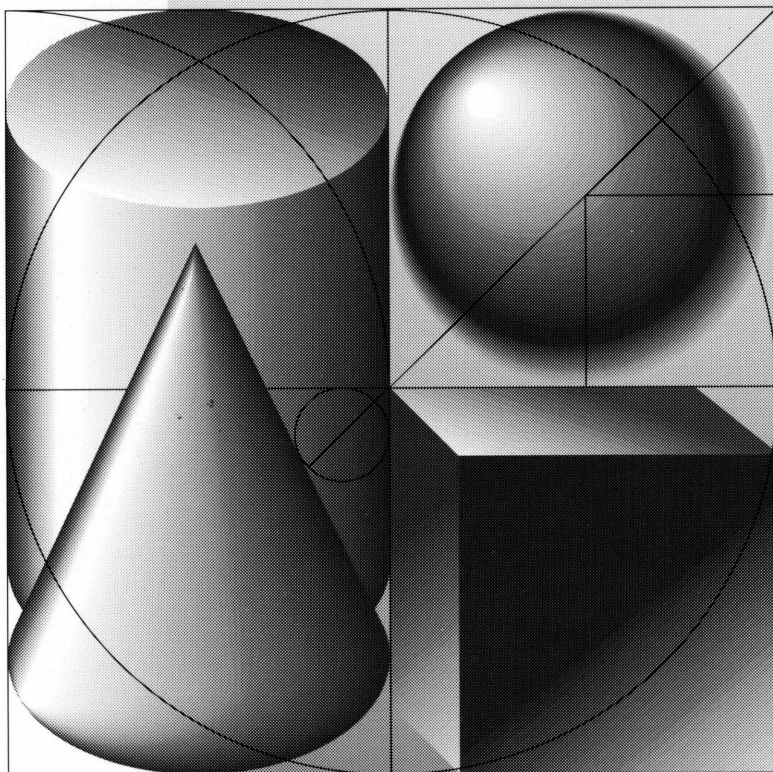




LocalTalk™ PC Card and Driver Preliminary Notes

APDA #M7055



AppleTalk PC Card and Driver

Preliminary Note

Final Draft
3 August, 1987
Project No. PMP017-01

Communications and Networking
Apple Technical Publications

APPLE COMPUTER, INC.

This manual is copyrighted by Apple, with all rights reserved. Under the copyright laws, this manual may not be copied, in whole or in part, without the written consent of Apple Computer, Inc. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased may be sold, given or lent to another person. Under the law, copying includes translating into another language.

© Apple Computer, Inc., 1986, 1987
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Apple, the Apple logo, AppleTalk, and LaserWriter are registered trademarks of Apple Computer, Inc. Macintosh and AppleShare are trademarks of Apple Computer, Inc.

IBM, IBM PC, IBM AT, and IBM XT are registered trademarks of International Business Machines Corporation.

Compaq is a registered trademark of Compaq Computer Corporation.

AT&T is a registered trademark of AT&T.

Microsoft is a registered trademark of Microsoft Corporation.

MultiMate is a trademark of Multimate International Corporation.

Simultaneously published in the United States and Canada.

NOTICE

The information in this document reflects the current state of the product.

Every effort has been made to verify the accuracy of this information;

however, it is subject to change. Preliminary Notes are released in this

form to provide the development community with essential information in

order to work on compatible third-party products.

Table of Contents

Page

i Table of Contents

v List of Figures

vii List of Tables

Preface

viii What These Notes Cover
viii Required Knowledge
viii Hardware and Software Requirements

Chapter 1. Introduction to the AppleTalk PC Card and Driver

1 Introduction
1 Overview
3 The AppleTalk PC Card
3 DIP Switch Settings
5 Configuration Options
6 Installing the AppleTalk PC Card
6 Installing the AppleTalk PC Driver
7 Memory Requirements
8 Setting Configuration Options

Chapter 2. AppleTalk PC Driver

11 Introduction
11 Data Definitions
12 Implementing AppleTalk Protocols
13 Internal Driver Limitations
13 Calling Conventions
14 Completion Routines
15 Protocol Handlers and Socket Listeners
15 Completion Routines and High-Level Languages
17 Protocol Parameter Structures
18 Conventions
18 Command Codes
20 Error Codes

Chapter 3. General Driver Commands

23	Introduction
23	Protocol Parameter Structures
24	Address Block
24	InfoParams
26	ATInit (\$1)
28	ATKill (\$2)
30	ATGetNetInfo (\$3)
33	ATErrorStatus
36	Timer Commands
38	ATGetClockTicks (\$4)
40	ATStartTimer (\$5)
42	ATResetTimer (\$6)
44	ATCancelTimer (\$7)

Chapter 4. Link Access Protocol (LAP) Commands

47	Introduction
49	LAPInstall (\$10)
51	LAPRemove (\$11)
53	LAPWrite (\$12)
55	LAPRead (\$13)
57	LAPCancel (\$14)

Chapter 5. Datagram Delivery Protocol (DDP) Commands

59	Introduction
62	DDPOpenSkt (\$20)
64	DDPCloseSkt (\$21)
66	DDPWrite (\$22)
68	DDPRead (\$23)
70	DDPCancel (\$24)

Chapter 6. Name Binding Protocol (NBP) Commands

73	Introduction
73	Protocol Parameter Structures
77	NBPRegister (\$30)
79	NBPRemove (\$31)
81	NBPLookup (\$32)
83	NBPConfirm (\$33)
85	NBPCancel (\$34)

Chapter 7. Zone Information Protocol (ZIP) Commands

87	Introduction
87	Protocol Parameter Structures
89	ZIPGetZoneList (\$35)
91	ZIPGetMyZone (\$36)

Chapter 8. AppleTalk Transaction Protocol (ATP) Commands

93	Introduction
93	Protocol Parameter Structures
96	ATPOpenSocket (\$40)
98	ATPCloseSocket (\$41)
99	ATPSendRequest (\$42)
102	ATPGetRequest (\$43)
104	ATPSendResponse (\$44)
107	ATPAddResponse (\$45)
109	ATPCancelTrans (\$46)
111	ATPCancelResponse (\$47)
113	ATPCancelRequest (\$48)
115	ATPRespond (\$49)

Chapter 9. Printer Access Protocol (PAP) Commands

119	Introduction
120	Protocol Parameter Structures
122	PAP Commands for the Workstation
122	PAPRead (\$72)
124	PAPWrite (\$73)
126	PAPCancel (\$7B)
128	PAP Workstation Commands
128	PAPOpen (\$70)
130	PAPClose (\$71)
132	PAPStatus (\$74)
134	PAPXStatus (\$7C)

Chapter 10. AppleTalk Session Protocol (ASP) Commands

137	Introduction
137	Protocol Parameter Structures
139	ASP Commands for the Workstation
139	ASPGetParms (\$50)
140	ASPCloseSession (\$51)
141	ASPCancel (\$52)
142	ASPGetStatus (\$5D)
143	ASPOpenSession (\$5E)
144	ASPCommand (\$5F)
146	ASPWrite (\$60)
148	ASPGetAttention (\$61)

Appendix A. C and the AppleTalk Driver

- A-1 Introduction
- A-1 Calling the Driver
- A-2 Converting Data Types
 - A-2 Converting a Pascal String to a C String
 - A-2 Converting an Entity Name to Three C Strings
 - A-3 Converting Three C Strings to an Entity Name
 - A-3 Fetching an Entity Name From the Buffer
- A-4 Parameter Blocks and 8086 Memory Models
- A-4 Examples of Data Structures

List of Figures

Page	Fig.	Title
2	1-1	Example of Using an AppleTalk PC Card
3	1-2	Overview of the AppleTalk PC Card and Driver
4	1-3	DIP Switches
16	2-1	State of the Stack
23	3-1	Parameter Block for General Driver Commands
24	3-2	Address Block
25	3-3	InfoParams Parameter Block
27	3-4	ATInit Parameter Block
29	3-5	ATKill Parameter Block
32	3-6	ATGetNetInfo Parameter Block
34	3-7	ATErrorStatus Parameter Block
37	3-8	General Parameter Block for Timer Commands
39	3-9	ATGetClockTicks Parameter Block
41	3-10	ATStartTimer Parameter Block
43	3-11	ATResetTimer Parameter Block
45	3-12	ATCancel Timer Parameter Block
48	4-1	General Parameter Block for Link Access Protocol (LAP) Commands
50	4-2	LAPInstall Parameter Block
52	4-3	LAPRemove Parameter Block
54	4-4	LAPWrite Parameter Block
56	4-5	LAPRead Parameter Block
58	4-6	LAPCancel Parameter Block
61	5-1	General Parameter Block for Datagram Delivery Protocol (DDP) Commands
63	5-2	DDPOpenSkt Parameter Block
65	5-3	DDPCloseSkt Parameter Block
67	5-4	DDPWrite Parameter Block
69	5-5	DDPRead Parameter Block
71	5-6	DDPCancel Parameter Block
74	6-1	NBPTuple Parameter Block
75	6-2	NBPTabEntry Parameter Block
76	6-3	General Parameter Block for Name Binding Protocol (NBP) Commands
78	6-4	NBPRegister Parameter Block
80	6-5	NBPRemove Parameter Block
82	6-6	NBPLookup Parameter Block
84	6-7	NBPConfirm Parameter Block
86	6-8	NBPCancel Parameter Block
88	7-1	General Parameter Block for Zone Information Protocol (ZIP) Commands
90	7-2	ZIPGetZoneList Parameter Block
92	7-3	ZIPGetMyZone Parameter Block

AppleTalk PC Card and Driver Preliminary Note

94	8-1	BDS Entry Format
95	8-2	General Parameter Block for AppleTalk Transaction Protocol (ATP) Commands
97	8-3	ATPOpenSocket Parameter Block
98	8-4	ATPCloseSocket Parameter Block
101	8-5	ATPSendRequest Parameter Block
103	8-6	ATPGetRequest Parameter Block
106	8-7	ATPSendResponse Parameter Block
108	8-8	ATPAddResponse Parameter Block
110	8-9	ATPCancelTrans Parameter Block
112	8-10	ATPCancelResponse Parameter Block
114	8-11	ATPCancelRequest Parameter Block
117	8-12	ATPRespond Parameter Block
120	9-1	Parameter Block for Status Record
121	9-2	General Parameter Block for Printer Access Protocol (PAP) Commands
123	9-3	PAPRead Parameter Block
125	9-4	PAPWrite Parameter Block
127	9-5	PAPCancel Parameter Block
129	9-6	PAPOpen Parameter Block
131	9-7	PAPClose Parameter Block
133	9-8	PAPStatus Parameter Block
135	9-9	PAPXStatus Parameter Block
138	10-1	Parameter Block for ASP Workstation Commands
139	10-2	ASPGetParms Parameter Block
140	10-3	ASPCloseSession Workstation Parameter Block
141	10-4	ASPCancel Workstation Parameter Block
142	10-5	ASPGetStatus Parameter Block
143	10-6	ASPOpenSession Parameter Block
145	10-7	ASPCommand Parameter Block
147	10-8	ASPWrite Parameter Block
148	10-9	ASPGetAttention Parameter Block

List of Tables

Page	Tbl.	Title
4	1-1	AppleTalk PC Card DIP Switches
5	1-2	Factory-Preset Switch Settings
5	1-3	Configuration Options
8	1-4	Memory Requirements
9	1-5	Configuration Parameters
12	2-1	Miscellaneous Data Definitions
19	2-2	Command Codes
21	2-3	Error Codes
30	3-1	Configuration Fields
47	4-1	Valid LAP Type Field Values
59	5-1	Valid DDP Type Field Values
60	5-2	Valid Socket Numbers

Preface

What These Notes Cover

These notes are written for developers and describe how to write a program that lets an IBM (or compatible) Personal Computer communicate with an AppleTalk® PC Card. The Card lets the user include an IBM (or compatible) Personal Computer in an AppleTalk network and use an Apple LaserWriter and an AppleShare File Server.

These notes include installation and configuration considerations, an overview of the card and driver, a detailed description of the programming interface of the AppleTalk PC driver, and a sample program in C.

In this document, the AppleTalk PC Card is referred to as “the Card.” IBM PC-compatible computers include the IBM PC, IBM XT, IBM AT, IBM PS/2 Model 30, Compaq, AT&T, and other such computers. In these notes, the computer for which the software is being written is referred to as “the PC.”

Required Knowledge

Before starting, the software writer should have a working knowledge of AppleTalk® network protocols (LAP, DDP, NBP, ZIP, ATP, PAP, and ASP). In addition, the writer should be familiar with the IBM PC or PC-compatible computers, DOS and device drivers, and the use of software interrupts to access system services (such as the BIOS and DOS). For further information, refer to the Apple Computer publication *Inside AppleTalk*, available through APDA (Apple Programmer's and Developer's Association).

Hardware and Software Requirements

The minimum PC configuration required for the AppleTalk PC Card is 256K of RAM with two disk drives. These notes assume the use of a PC with a hard disk; however, a system with floppy disks only can be used. There must also be an expansion slot available for the AppleTalk PC card. Approximately 40K of memory is required for the AppleTalk PC Card Driver, Version 2.0.

Also required is an AppleTalk connector (a connection box) and an AppleTalk cable (the 2-meter cable that attaches one connection box to another on the AppleTalk Personal Network). If the network requires other AppleTalk products, such as a cable extender or a custom-wiring kit, refer to the *AppleTalk Personal Network* manual.

Important: The AppleTalk plug connection for the IBM computer is a D-shaped plug with nine pins. Ensure that the correct plug is available (other AppleTalk plugs are round with only eight pins and will not fit the AppleTalk PC Card in the IBM PC or PC-compatible computer).

Chapter 1

Introduction to the AppleTalk PC Card and Driver

Introduction

This chapter contains general information about the AppleTalk PC Card and driver software. The following information is given to prepare the Card for operation:

- an overview of the Card and driver
- a description of the Card and switch settings
- instructions for installing the Card
- instructions for configuring and installing the driver

See Chapter 2 for information on writing software that uses the AppleTalk PC driver, including data definitions, calling conventions, syntax conventions, and command codes.

Overview

The AppleTalk PC Card provides an intelligent interface between an IBM Personal Computer and an AppleTalk Personal Network. It is the “AppleTalk engine” for the IBM PC and PC-compatible computers. With this Card, software can be developed to allow these computers to communicate with other AppleTalk-compatible devices (such as the Apple® LaserWriter printer or AppleShare file servers) over the AppleTalk network, as shown in Figure 1-1.

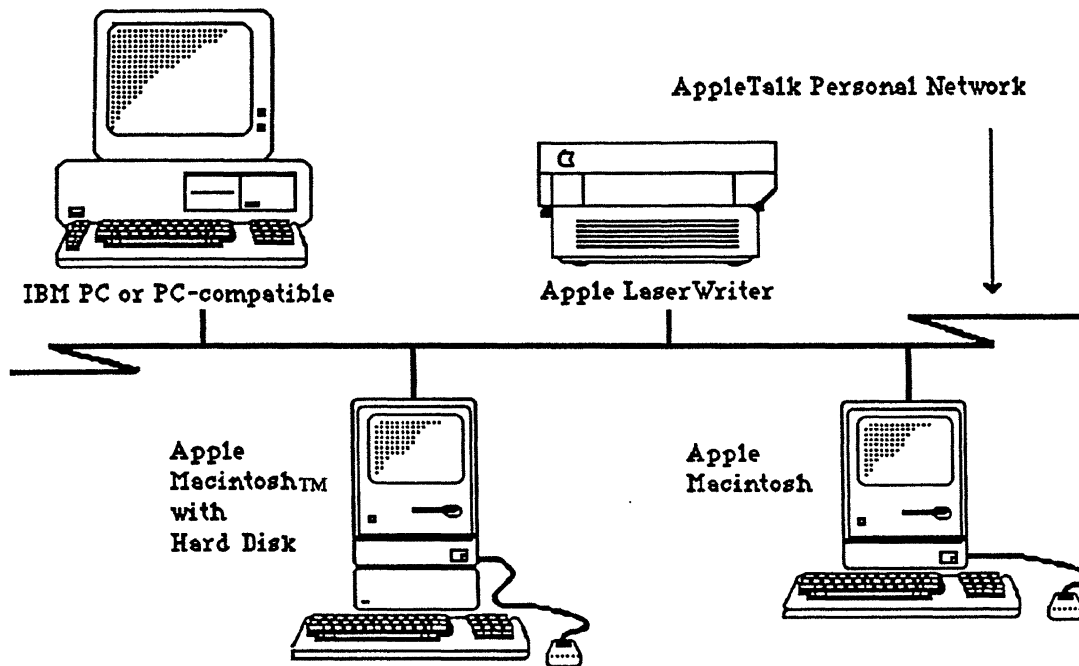


Figure 1-1. Example of Using an AppleTalk PC Card

The Card contains firmware to control the link between the AppleTalk network and the PC. An application that will use the network must set up appropriate parameter blocks for the desired commands, and then call the driver (described in Chapters 3 and later chapters). The AppleTalk PC Driver software responds to the commands by implementing some of the AppleTalk protocols on the Card and some of the protocols in the Driver. The protocols are then sent over the network. Figure 1-2 illustrates the data flow of this process.

Note: As shown in Figure 1-2, the AppleTalk PC Card resides in a slot in the IBM PC (or PC-compatible computer). The AppleTalk PC Driver resides in the memory of the PC.

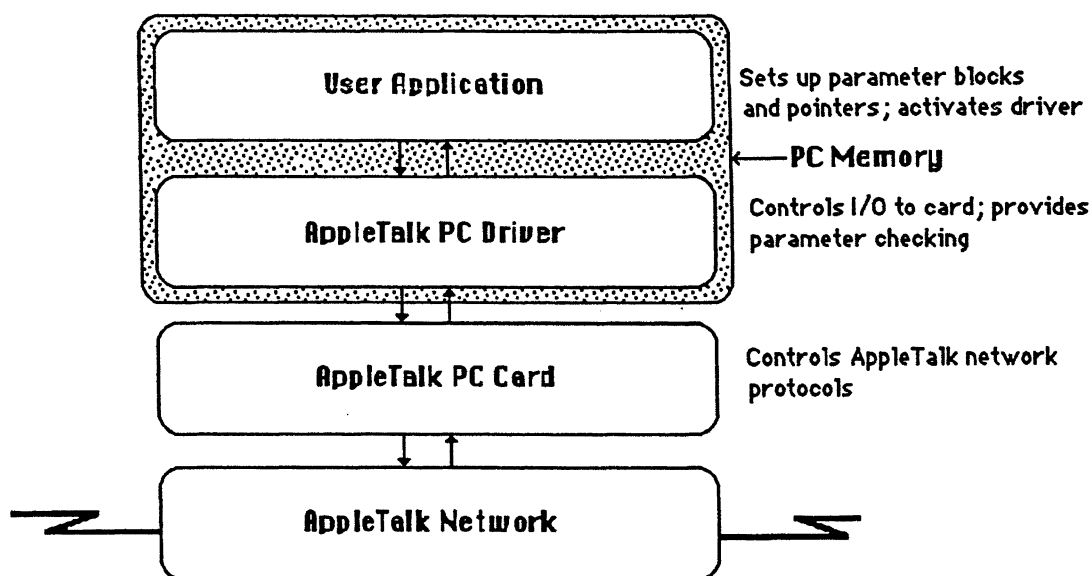


Figure 1-2. Overview of the AppleTalk PC Card and Driver

The AppleTalk PC Card

The Card conforms to the PC-bus specifications. The Card can be placed in any slot within a PC. (However, slot 8 of the IBM XT cannot be used because critical signals are missing from the card sockets of this slot.) The card is a short card, which means that it may be used in the slots normally reserved for serial-communication cards. Before installing the AppleTalk PC Card, examine the Card's layout and switch settings, and set the Card's switches for operation in the PC as described in the next sections.

DIP Switch Settings

DIP switches on the Card allow it to respond to one of two Card address ranges, and to use one of two DMA channels. Figure 1-3 shows the DIP switches on the Card.

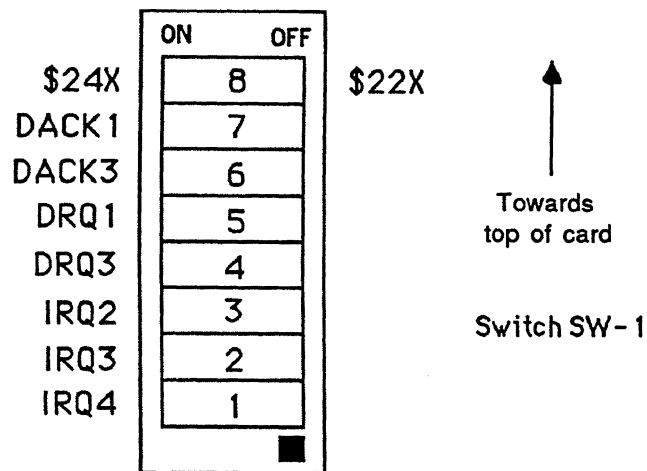


Figure 1-3. DIP Switches

The range of settings and a brief description of each are shown in Table 1-1 .

Table 1-1. AppleTalk PC Card DIP Switches

Switch	Description	Function
1, 2, and 3	IRQ4, IRQ3, IRQ2	"Interrupt Request" priority-level selection
The Interrupt Priority is not used. All three switches must be off for Version 2.0 of the Driver.		
4 and 6	DRQ3, DACK3	"DMA Request" and "DMA Acknowledge" on DMA channel 3
Both of these switches must be on (and both switches 5 and 7 must be off) for the Card to communicate with the PC via DMA channel 3.		
5 and 7	DRQ1, DACK1	"DMA Request" and "DMA Acknowledge" on DMA channel 1
Both of these switches must be on (and both switches 4 and 6 must be off) for the Card to communicate with the PC via DMA channel 1.		
8	\$24x/\$22x	I/O address range of Card
If this switch is on, the Card will respond to control signals sent to I/O addresses in the range \$240 to \$247. If this switch is off, the Card will respond to control signals sent to I/O addresses in the range \$220 to \$227.		

The AppleTalk PC Card is shipped with the switch settings shown in Table 1-2. With these settings, the Card is configured for DMA channel 1 and addresses \$240 through \$247.

Table 1-2. Factory-Preset Switch Settings

Switch	1	2	3	4	5	6	7	8
On	-	-	-	-	x	-	x	x
Off	x	x	x	x	-	x	-	-

Note: The default settings on the Card conflict with the COM2 port on the PC. If both COM ports are used (such as when using both a serial port mouse and a modem), the PC Card cannot be used.

If other add-on cards are installed in the PC, there may be hardware conflicts that compete for these settings. (Refer to the manuals for the other Cards to determine any settings that may conflict.) Change the settings on the AppleTalk PC Card so that the operation of the Card will not be affected. If the switches need to be reset, use a piece of wire, such as a straightened paperclip or a small screwdriver, to move the switches gently to their proper settings. NEVER use a pencil to move the switches.

Configuration Options

There are several options available for configuring the Card and driver to operate in a particular individual PC. Table 1-3 lists these options.

Table 1-3. Configuration Options

Option	Configures
Card address	The Card I/O address
DMA channel	The DMA channel used by the Card
Access interrupt	The software interrupt used by an application in the PC to access the driver (driver requests are set at interrupt 0x60 by default)
Memory	The size of the driver memory pool

To configure the Card to any settings other than those shipped from the factory, change the DIP switches before installing the Card in the PC. Configure the driver during installation by specifying the new settings for the options in Table 1-3. Configuration and installation for both the Card and driver are described next in this chapter.

Installing the AppleTalk PC Card

On any computer except the IBM XT, install the AppleTalk PC Card into any one of the PC's slots. On the XT, do not use slot 8 (the rightmost short slot) because of differences in signals available for that slot. Install the Card in the PC. (Refer to the computer's manual for information on how to remove the casing and install the card.)

Warning: Be sure to touch the computer's back metal frame to achieve grounding. Because the Card uses CMOS components, grounding will prevent static electricity from possibly damaging sensitive internal components or damaging the Card. Pay strict attention to the warning messages inside the computer. Various electronic parts may hold high voltages even when the power cord is disconnected.

After installing the AppleTalk PC Card, attach the PC to an AppleTalk network. The exact connection of the 2-meter cable to the network depends on the location of the PC in relation to other network devices (for instance, between other devices or at the end of the network). Refer to the *AppleTalk Personal Network* manual for additional information on networking.

With the Card installed, proceed to install the AppleTalk PC driver as described in the next section.

Installing the AppleTalk PC Driver

Two disks are provided with the AppleTalk PC Card: a Startup disk and an Applications disk. Follow these steps to copy all the necessary software automatically from the two disks to the hard disk or work disks (floppy disks) and to install the driver:

1. Boot the machine from the boot disk.
2. Insert the Startup disk in drive A.
3. At the DOS prompt, type "A:INSTALL" and press the Enter key. Detailed instructions will be displayed on the screen for installation on either the hard disk or floppy-based system.
4. When installation is complete, store the Startup and Applications disks in a safe place.

For hard disks, the installation program places the driver (ATALK.EXE) in the root directory, and the applications in the directory \LWDIR. The AUTOEXEC.BAT file is modified to include \LWDIR in the PATH statement. For example, if the path statement is "PATH=C:\BIN", the path statement will appear as follows:

```
PATH=C:\BIN;C:\LWDIR
```

This change should not affect the operation of other programs on the hard disk, and does allow the LaserWriter applications that come with the Card to be located regardless of the default directory used.

The installation program also adds the line "ATALK" to the AUTOEXEC.BAT file which loads and starts the driver at boot time.

Memory Requirements

The AppleTalk PC driver has a memory buffer pool that is used for processing client user requests. The default pool size is approximately 2K (2048) bytes, which is available for use in processing driver calls and should be adequate for applications doing simple processing. However, applications with multiple PAP or ASP connections, multiple outstanding driver requests, or both can exceed the driver's default pool size. Applications should be aware that the driver may return a NO_MEM_ERROR to an otherwise good command, and then react accordingly.

Table 1-4 lists driver entities and their memory requirements. This table can be used to estimate the minimum driver buffer required to support a given application and can be changed by setting the configuration option for memory accordingly.

Table 1-4. Memory Requirements

Bytes	Driver Entity
16	LAP Read
48	Install a protocol handler
16	DDP Read
48	Install a socket handler
16	Dispatch an AT Start Timer
32	NBP Register, NBP Lookup or NBP Confirm
32	ATP Send Request
32	ATP Open Socket with a filter
32	ATP Get Request
32	ZIP GetMyZone or ZIP GetZoneList
112	PAP Status by name during the name lookup
280	PAP Status by address or after name lookup
368	PAP Open by name during name lookup
352	PAP Open by address or after name lookup
384	Each Open PAP connection
32	PAP Read
240	PAP Init during name registration
160	Maintain an open PAP server
673	Start each connection on a PAP server
384	Maintain a server PAP connection
256	Maintain an ASP Workstation
160	Each outstanding command
196	Each outstanding write
384	Start an ASP session

Setting Configuration Options

Once the Card has been installed, the driver is accessed through a software interrupt. Using this interrupt, a programmer can gain access to the various network protocols as defined by Apple Computer, Inc. The configuration options can be changed for the driver to give additional flexibility. However, the switch settings may have to be changed on the Card as well as in other application programs to recognize these new options.

Table 1-5 lists the configuration parameters that change the default values of system parameters used by the driver. These options are specified in lowercase following the driver name (ATALK) in the AUTOEXEC.BAT file.

Table 1-5. Configuration Parameters

Parameter	Default	Function
/address=xxx	240 (Hex)	Sets the card I/O address
/dma=x	1	Sets the DMA channel that the card uses
/int=xx	60 (Hex)	Sets the software interrupt an application used to access the driver
/mem=xxxxxx	8192 (Dec) (8K)	Sets the size of the driver memory pool (in bytes)
/cardint=x	3 (Dec)	Sets the interrupt number the card uses to access the processor (unused by driver, Version 2.0)

If these values do not conflict with other expansion cards already installed in the PC, they need not be specified. If the values must be changed, please consult the appropriate technical reference manuals for the PC.

Chapter 2

AppleTalk PC Driver

Introduction

The AppleTalk PC driver is a general-purpose interface to AppleTalk protocols that can be called by DOS applications in order to use the AppleTalk network. The interface described in these notes is for the AppleTalk PC Card; other cards may conform to all or part of the interface.

This chapter contains general information about the AppleTalk PC driver, including

- data definitions
- using AppleTalk protocols
- calling conventions necessary to write software using the AppleTalk PC driver
- protocol parameter structures and syntax conventions
- command codes and error returns

Data Definitions

The following conventions are used in discussing the AppleTalk PC driver:

- Hexadecimal numbers are represented as \$xxx; all other numbers are decimal.
- All character strings passed to the AppleTalk PC driver are “Pascal-style” strings. These strings consist of one byte that indicates the length of the string (exclusive of the length byte) followed by the characters (bytes) that make up the string.
- The following words describe data objects in the various parameter structures of the driver commands:

<u>Abbreviation</u>	<u>Size (in bytes)</u>
byte	1
word	2
dword	4
char[<i>n</i>]	<i>n</i>

Whenever an application refers to an address, it is stored as a two-word *segment:offset*, with the *offset* stored first. Word values are stored in standard 8086 order, with the least-significant byte first. The exceptions are the network number and transaction ID, which are stored with the most-significant byte first. The mnemonics and parameter field labels are used to describe the interface to and behavior of the various commands in Chapter 3, “General Driver Commands.”

All registers are preserved except for AX, which returns the error code from the driver. The application also uses registers DS:BX for the *segment:offset* of parameter structures passed to and from the driver. Registers AH, CX, DS:BX, and ES:DX are used with handlers and listeners.

- Table 2-1 lists miscellaneous data definitions. Their usage is explained in relevant sections of the following chapters on driver commands.

Table 2-1. Miscellaneous Data Definitions

Name	Value	Used for
Async mask	8000H	Having any driver command execute asynchronously
Interrupt mask	4000H	Interrupt routines making driver calls
XObit	20H	ATP - indicates exactly Once transaction
EOMbit	10H	ATP - indicates End Of Message
STSbit	8H	ATP - asks requester to Send Transaction Status
NoRelease	2H	ATP - keeps ATP from sending release on XO transactions
ChkSum	1H	ATP - calculates checksum on outgoing packets
InfiniteRetry	0H	ATP - specifies infinite number of retries for command

Implementing AppleTalk Protocols

This section describes the PC implementations of the AppleTalk protocols. (For a complete description of AppleTalk protocols, refer to *Inside AppleTalk*.)

Because of differences between the PC architecture and the network hardware or the operating environments of the PC and the Macintosh™, there are two fundamental factors in the conventions that affect implementation:

- Two lower-level protocols, LAP and DDP, have provisions for “user-installed” protocol handlers (or socket listeners). Therefore, when the protocol handler (socket listener) is installed, the incoming network packets will be received and the application will be notified by the execution of code that is supplied. These routines provide a flexible, although low-level, communication service.
- All AppleTalk network commands may be performed asynchronously. The completion of these commands can be detected by the calling application through monitoring a status word through the execution of a completion routine, or both. Like the listeners, completion routines are “user-supplied” pieces of code that are called by the driver upon the completion of some event.

The details of the constraints on both listeners and completion functions are described in the section, “Calling Conventions.”

Internal Driver Limitations

Two protocols defined as part of AppleTalk are not covered in this document: the Routing Table Maintenance Protocol (RTMP) and the Echo Protocol (EP). These protocols do not have any user calls, so there is no need to specify an interface. However, it is assumed that any AppleTalk PC driver written to the specifications detailed in these notes will properly implement these protocols as described in *Inside AppleTalk*.

All other driver limitations are based either on the size of an internal memory pool from which the driver allocates space or on intrinsic AppleTalk limitations. Examples of this include the number of sockets open simultaneously, the number of current ATP requests, and the number of PAP connections. The default size of this pool is about 2K, but can be changed via the driver's "/mem" option (see Table 1-4).

Calling Conventions

The AppleTalk PC driver is accessed by issuing a software interrupt with registers DS:BX pointing to a protocol parameter block. Before the software interrupt is issued, the command code and the relevant fields of the parameter block should be filled according to the specification for that command. Fields that are part of the parameter block but that are not used for a call should be set to zero. The user should not depend on the values in any fields of the parameter block that are not explicitly defined.

If the caller wants the command to be executed asynchronously, the high bit of the command word in the parameter block should be set. If the high bit is not set, the command will be executed synchronously.

Note: If the AppleTalk PC driver has not been installed, or if the driver is using an Access Interrupt different from the one expected by the caller, executing the software interrupt to access the driver may have unexpected results. To check if the driver is installed, look for the word "AppleTalk" in memory by starting at 16 bytes before the location pointed to by the Access Interrupt vector. For example, use large-memory-model C and Access Interrupt=0x60 as follows:

```
if ( strcmp ("AppleTalk", * (char **) (0x60*4) - 16, 9) )
{
    error ("Driver not loaded!");
}
```

When the command completes, the driver fills in the status word in the parameter block with the proper return value (zero means no error occurred). Other fields in the parameter structure may be filled by the driver, depending on the command. Unused fields can also be modified by the driver. The byte immediately following the word "AppleTalk" is the minor version number. The next byte is the major version number, and the third byte is the driver type. For the AppleTalk driver, the type is a "1". Version 2.0 is stored as 200H.

If the command is executed *synchronously*, upon return from the driver (and completion of the command) the register AX will hold the same value as the status word in the parameter block. If the command is executed *asynchronously* when the command request returns from the driver, then AX will be zero if no error has been encountered yet and the command is still in progress. If an error has occurred, AX will hold the same value as the status word and the command will be completed. The caller should not modify any parameter structure supplied to the AppleTalk PC driver until the command has been completed. The application may detect completion of an asynchronous command by monitoring the status word of the parameter block. While the command is in process, that word will have a positive value. Upon completion, the status word will be either zero if no error has occurred, or filled with the relevant negative error code if an error has occurred.

To request a driver service from an interrupt routine, set the interrupt mask bit (4000H) on the command word. The request will be internally queued and the driver will process it at the next opportunity. A NO_MEM_ERROR will be returned if the internal queue is full.

Completion Routines

It is possible to specify a completion routine with all command requests. The AppleTalk PC driver will call this routine when the command completes, by passing the address of the parameter block. The conventions for completion routines are outlined in the next section. When the driver calls the completion routine, the driver will have filled in the status word with the appropriate error return.

Completion routines are always called. Such things as cancelling requests or closing a socket result in all pending completion routines being called with the appropriate cancel code (ATP_Cancelled, DDP_Cancelled, and so forth).

The following conventions are true for completion routines:

- All routines are given control through a FAR CALL, and should exit a FAR RET.
- The routines are called at interrupt time, so they should not make DOS function calls (int 21H) or BIOS calls because that code may not be re-entrant.
- The routines will be called with a minimum of 512 bytes of stack, and should *not* modify the stack environment.
- Any registers other than SS and SP may be altered upon return to the driver.
- The routines are called with DS:BX pointing to the parameter structure that was passed when the call was initiated, and ES=DS.
- The routines are called with interrupts enabled. It is acceptable to make another command request from the completion routines (for example, to issue a ATPGetRequest). However, a command cannot be called synchronously that requires use of the network (as opposed to those which just involve manipulation of internal driver tables). Calling a command asynchronously is allowed.

Note: There are two exceptions to this convention—the LAPRead and DDPRead commands, which are described in Chapter 3, "General Driver Commands."

Protocol Handlers and Socket Listeners

The requirements of listeners are similar to those of completion routines (as defined in the preceding section). However, because the listeners (DDP and LAP protocols) work at a lower level, there are important differences. For more information, refer to Chapter 4 on LAP commands and Chapter 5 on DDP commands.

The following conventions are true of handlers and listeners (LAP and DDP):

- All routines are given control through a FAR CALL and should exit with a FAR RET.
- The routines are called at interrupt time, so they must not make DOS function calls (int 21H) or BIOS calls because that code may not be re-entrant.
- The routines will be called with approximately 512 bytes of stack, and should *not* modify the stack environment.
- Only registers SS and SP must not be altered upon return to the driver. All registers are preserved except for AX, which returns the error code from the driver. Register DS:BX is used for the *segment:offset* of parameter structures passed to and from the driver.
- The routines that the user supplies when the listener is installed are called at the time when a packet is received. The routines are given information regarding the packet's header, and may choose at that time to receive or throw away the actual data in the packet. Therefore, when the routines return and the data is to be saved, the listener must supply the driver with a buffer address for the data, as well as the address of code to be executed when the data reception (or transfer) is completed.

Thus, listener routines must have two parts: one that handles the packet header and chooses what to do with the data (the "first-half"), and another that is called after the data is in place (the "second-half").

- The first-half routine is called with interrupts disabled. It should *not* enable interrupts as the incoming data may be lost.
- Both types of routines are called with packet information in registers described in Table 2-1, and the first-half routines must set registers for the driver's use before returning to the driver.

Other information about parameters of these routines is included in relevant sections of the chapters on the various driver commands.

Completion Routines and High-Level Languages

Although this document describes an assembly language interface, it is possible to obtain the services of the AppleTalk PC driver by using higher-level languages (such as Pascal or C). Although it is impossible to satisfy the conventions of every different C implementation for the PC, the conventions used by the AppleTalk PC driver are designed to meet the requirements of many existing C implementations. This section discusses some considerations for writing completion routines in C; refer to Appendix A for a sample program written with the Microsoft C Compiler, version 3.0.

Completion routines are called as FAR procedures; that is, with a 4-byte *segment:offset* return address pushed on the stack. DS:BX points to both the parameter block associated with this completion, and ES=DS. To facilitate writing completion routines in high-level languages that pass parameters via the stack, DS:BX is also pushed onto the stack directly above the return address. Note that SS is the driver's stack segment which will not equal DS, as some small-memory-model languages require.

The diagram in Figure 2-1 shows the state of the stack at the onset of a completion routine.

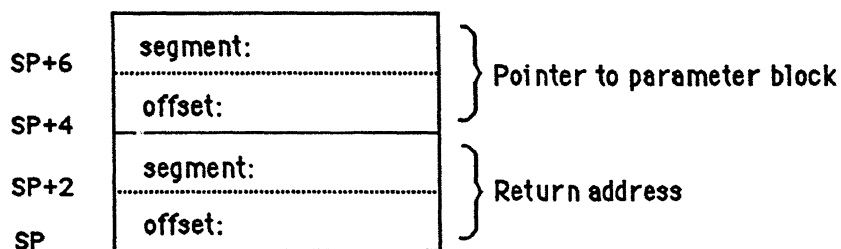


Figure 2-1. State of the Stack

Completion routines are called with interrupts enabled, except in the case of completion routines for LAPRead and DDPRead, in which the calls were made with bufptr=0.

Completion routines are allowed to make other driver calls. A user interrupt routine may call the driver if it has set the Interrupt Mask and the Async Mask. Doing this will allow the driver to queue up the request and execute it at the next opportunity. The driver will enable interrupts. Therefore, the interrupt routine must protect against this.

Because completion routines are called during interrupt processing, they should be short and quickly completed. This practice is especially important for routines that are called with interrupts disabled.

Protocol Parameter Structures

Both the parameter structures used by the AppleTalk PC driver and the control blocks for the various commands are described in this note by using parameter blocks. Actual assembly language declarations are also provided. The structure descriptions have the format

StructureName	fieldtype	fieldLabel;	description
---------------	-----------	-------------	-------------

where the following hold true:

fieldtype = the size (or StructureName, for imbedded structures) of the field

fieldLabel = a name that is used in the driver command descriptions

description = a brief description of the use of that field

The first three fields in each protocol parameter structure include the following:

word	atd_command;	command code
word	atd_status;	status word/error return
dword	atd_compfun;	address of completion routine or 0

These fields are used in every call to the AppleTalk PC driver. The command codes defined in this chapter are the legal values for the atd_command field. Upon completion, the atd_status field will contain an error return. If the atd_compfun field is non-zero, the code pointed to by that field will be called with the registers set as described in the section "Completion Routines." Refer to the Error Codes at the end of this chapter for a list of currently defined error returns.

Conventions

This note uses the following syntax and layout conventions to describe each driver command:

Parameter Usage

<-- or -->	label	description
------------	-------	-------------

Error Returns

xxxxxxxxxx

The Command Name is a block of text that briefly describes the behavior of the command. The parameter block details which fields of the structure need to be input, as well as which fields are used for output. In the text, the arrow next to the field label indicates which fields are input (<-->) and which fields are output (<--); the description field gives a brief explanation of how the field is used for that call. Some fields may be used for both inputs and outputs on certain commands. Error Returns lists the valid error codes for this command. In addition, each command is illustrated with a sample parameter block, with arrows showing input (<-->) and output (<--).

In each chapter on driver commands for each protocol group, the introduction includes information common to all or most of the commands in that chapter.

The application can call any command asynchronously by setting the high bit in the command-code field of the parameter structure (see AsyncMask in Table 2-1). To detect the completion of asynchronous commands, the application should monitor the atd_status field. This field will be positive (high bit cleared) while the command is in progress, or set to zero or a negative number (high bit set) when the command completes.

All of the driver commands can accept a completion routine, which the driver calls when the command is done. The driver calls the completion routine with the registers DS:BX containing the *segment:offset* of the parameter structure (see the section on “Completion Routines and High-Level Languages” earlier in this chapter for more information).

The completion routine may examine the atd_status field of the parameter block to determine if any errors occurred.

Command Codes

This section describes the command codes that the application should use to call the driver. The application should place command codes in the atd_command field of the passed parameter block before issuing the interrupt. The command codes are listed in Table 2-2.

Table 2-2. Command Codes

Command Name	Command Number (in Hex)
ATInit	1
ATKill	2
ATGetNetInfo	3
ATGet ClockTicks	4
ATStartTimer	5
ATResetTimer	6
ATCancelTimer	7
LAPInstall	10
LAPRemove	11
LAPWrite	12
LAPRead	13
LAPCancel	14
DDPOpenSocket	20
DDPCloseSocket	21
DDPWrite	22
DDPRead	23
DDPCancel	24
NBPRegister	30
NBPRemove	31
NBPLookup	32
NBPConfirm	33
NBPCancel	34
ZIPGetZoneList	35
ZIPGetMyZone	36
ATPOpenSocket	40
ATPCloseSocket	41
ATPSendRequest	42
ATPGetRequest	43
ATPSendResponse	44
ATPAddResponse	45
ATPCancelTrans	46
ATPCancelResponse	47
ATPCancelRequest	48
ATPRespond	49

Continued on next page

Table 2-2. Command Codes Continued

Command Name	Command Number (in Hex)
ASPGetParms	50
ASPClose Session	51
ASPCancel	52
ASPGetStatus	5D
ASPOpenSession	5E
ASPCommand	5F
ASPWrite	60
ASPGetAttention	61
PAPOpen	70
PAPClose	71
PAPRead	72
PAPWrite	73
PAPStatus	74
PAPRegName	75
PAPRemName	76
PAPInit	77
PAPNewStatus	78
PAPGetNextJob	79
PAPKill	7A
PAPCancel	7B
PAPXStatus	7C

Error Codes

Table 2-3 lists the error codes returned by functions. Errors may be indicated in one of two ways. Upon return from the driver routine, a negative number may be returned in AX. For asynchronous calls, the result of the command will be indicated in the parameter block "status" word.

While the command is still executing, there will be a positive value (should be a 1) in the status field. When the command has finished, the status field will be set to either zero to indicate successful completion, or a negative number (high bit set) to indicate an error.

Table 2-3. Error Codes

Name	Value of AX		Description
	Decimal	Hex	
NOERR	0	0	No errors yet encountered
MAXCOLISERR	-1	FFFF	Too many collisions detected
MAXDEFERERR	-2	FFFE	32 deferrals encountered (ALAP)
LAP_LENERR	-30	FFE2	LAP data too long
LAP_TYPERR	-31	FFE1	Bad LAP type
TABFULLERR	-32	FFE0	No more LAP protocols available
LAP_NOTFND	-33	FFDF	LAP type not found in internal protocol table
LAP_CANCELLED	-34	FFDE	LAP command has been cancelled
DDP_SKTERR	-40	FFD8	Used unopen socket
DDP_LENERR	-41	FFD7	DDP data too long
NOBRDGERR	-42	FFD6	No bridge found
DDP_CANCELLED	-43	FFD5	DDP transaction cancelled
ATP_REQFAILED	-100	FF9C	No response received
ATP_NOSENDRESP	-101	FF9B	Original socket not found
ATP_BADSOCKET	-102	FF9A	ATP socket not found
ATP_NORELEASE	-103	FF99	No release received on ATP request
ATP_OVERFLOW	-104	FF98	Too much response data
ATP_CANCELLED	-105	FF97	Transaction cancelled
ATP_NO_IDS	-106	FF96	No more transaction IDs for ATP
NO_MEM_ERROR	-120	FF88	Not enough memory to perform requested operation
BAD_PARAMETER	-121	FF87	Illegal parameter or parameters
STACKERROR	-122	FF86	Too many levels of nesting to complete call
ATNOTINITIALIZED	-123	FF85	Driver not initialized
CALLNOTSUPPORTED	-124	FF84	Requested call not supported
HARDWARE_ERROR	-125	FF83	Unrecoverable hardware error
SOFTWARE_ERROR	-126	FF82	Internal software error. Reinitialize driver
MEMORY_CORRUPTED	-127	FF81	Internal memory pool corrupted. Reload driver
BAD_SYNC_CALL	-128	FF80	Synchronous command was called at a time when the driver cannot handle it

Continued on next page

Table 2-3. Error Codes Continued

Name	Value of AX		Description
	Decimal	Hex	
NBP_NEWSOCKET	-200	FF38	New entity address on confirm
NBP_NOCONFIRM	-201	FF37	Entity not found on confirm
NAME_IN_USE	-202	FF36	Name already in use
NBP_NO_ROOM	-203	FF35	Buffer overflow
BAD_NAME	-204	FF34	Illegal entity name
NBP_NOTFOUND	-205	FF33	Name not found
NBP_CANCELLED	-206	FF32	NBP command cancelled
NBP_NO_IDS	-207	FF31	No more IDs for NBP
TMR_NOTFOUND	-215	FF29	Timer structure not found
TMR_CANCELLED	-216	FF28	Timer command cancelled
PAP_BADCONNID	-300	FED4	Invalid connection reference number
PAP_NOCONNIDS	-301	FED3	No more connections available
PAP_DIED	-302	FED2	Connection died
PAP_LENERR	-303	FED1	Data length too large
PAP_WRITE_ACTIVE	-310	FECA	Write already active on stream
PAP_READ_ACTIVE	-311	FEC9	Read already active on stream
ASP_DIED	-401	FE6F	ASP session tickle timer expired
ASP_CANCELLED	-402	FE6E	ASP command was cancelled
ASP_BADCONNID	-403	FE6D	ASP session reference number does not exist

The following chapters provide detailed information on the various driver commands.

Chapter 3

General Driver Commands

Introduction

This chapter describes the general commands that the AppleTalk PC driver recognizes, as well as the parameter structures, required inputs, register usage conventions, and valid error returns for each command. Other commands are described in later chapters.

The general driver commands include

- ATInit
- ATKill
- ATGetNetInfo
- AppleTalk Timer commands

The commands described in this chapter provide a way to check the status of the driver (including network address information of the node), and to initialize and terminate network activity. In all cases, these calls will return zero (0 = NOERR) if the call is completed successfully, or a negative number if there is an error.

Protocol Parameter Structures

The application builds the parameter blocks, which activate the driver with a pointer to the parameter block. All of the information the driver needs is in the parameter block. Figure 3-1 gives an example of the general parameter block.

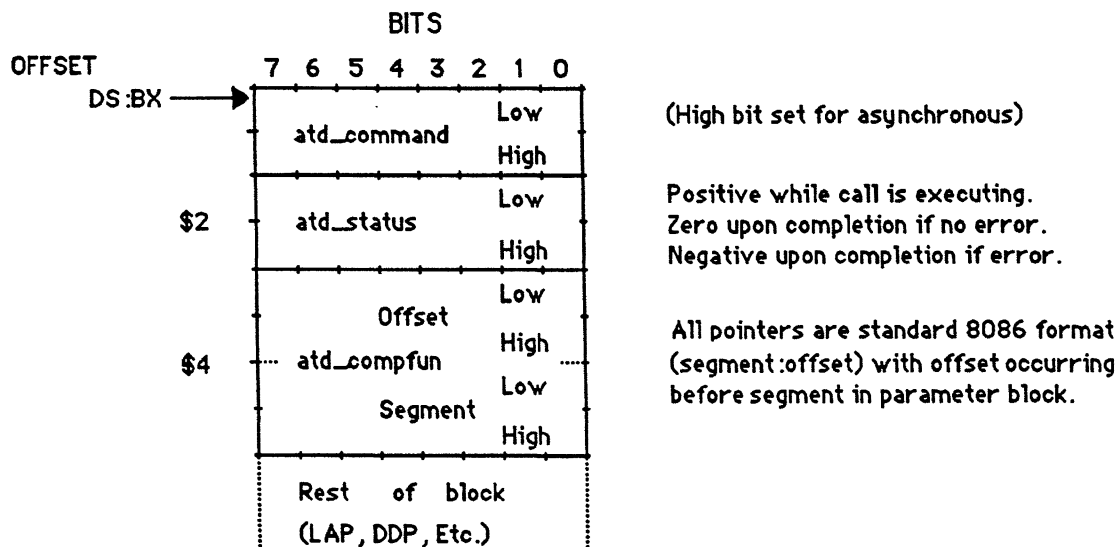


Figure 3-1. Parameter Block for General Driver Commands

Address Block

An AddrBlk is a structure describing the full network address of a socket. If the network field is zero, the driver assumes it is the local network, even if the local network has an assigned number. Many commands include this structure as an embedded structure within the other parameter structures.

Parameter usage for the address block is as follows:

AddrBlk		
word	network;	network number
byte	nodeid;	node number
byte	socket;	socket number

Figure 3-2 illustrates the address block.

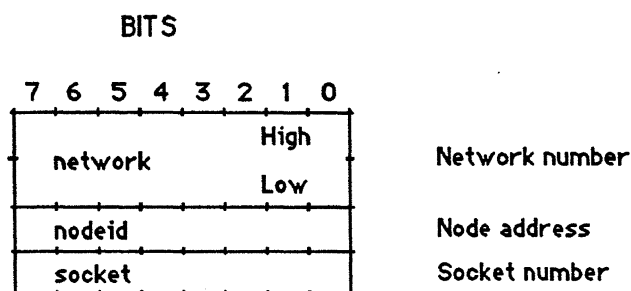


Figure 3-2. Address Block

Note: Any parameter that uses a network number or transaction ID are put in AppleTalk network ordering (high byte first, low byte last) and not in PC ordering.

InfoParams

The application uses the InfoParams structure in the interfaces to the general AppleTalk driver commands ATInit, ATKill, and ATGetNetInfo.

The following protocol parameter structure is used in those interfaces:

InfoParams		
word	atd_command;	command code
word	atd_status;	status word/error return
dword	atd_compfun;	completion routine or zero
word	inf_network;	network number or zero
byte	inf_nodeid;	current node number or zero
byte	inf_abridge;	node of a bridge or zero
word	inf_config;	configuration mask for driver
dword	inf_buffptr;	segment:offset of buffer for error info
word	inf_buffsize;	size of buffer

Figure 3-3 gives an example of the protocol parameter structure for InfoParams.

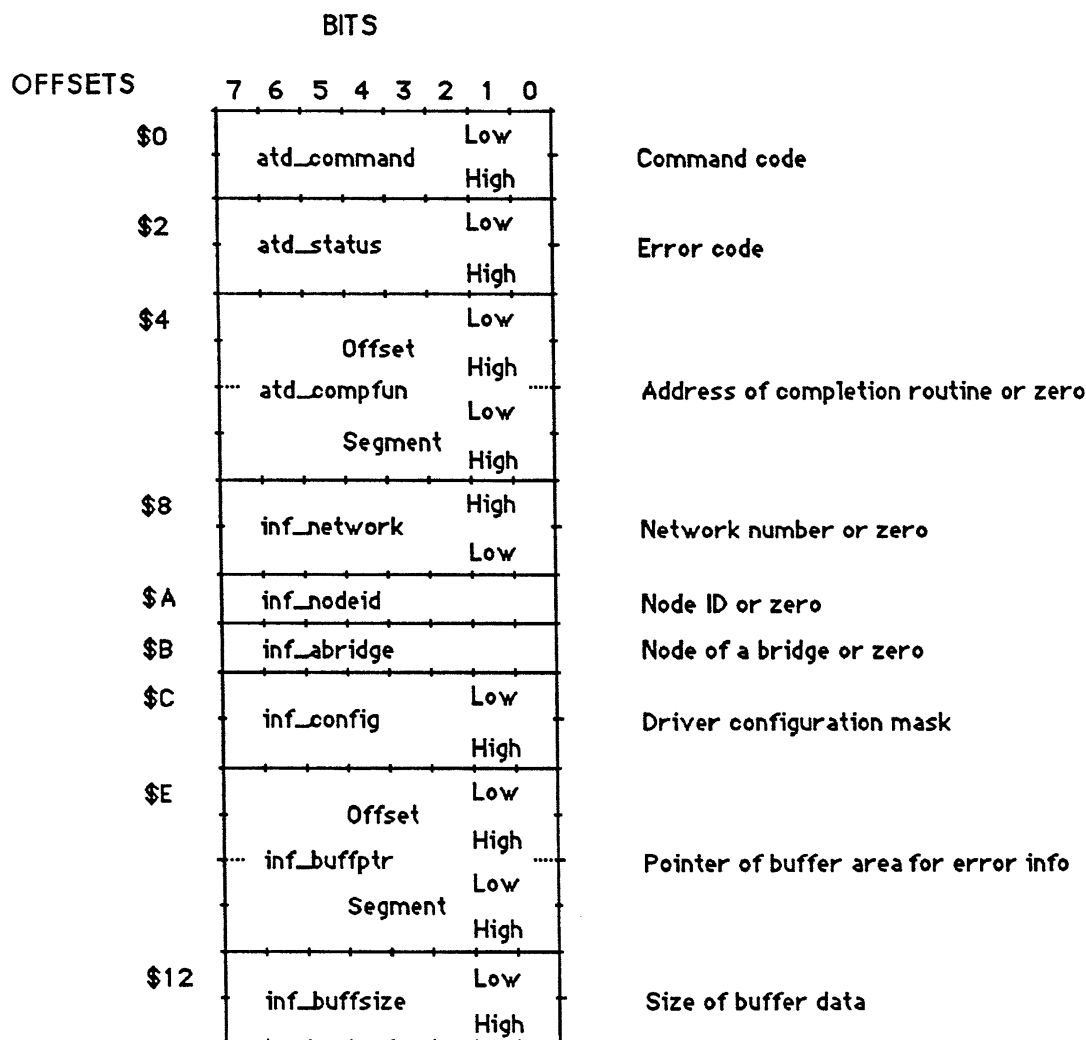


Figure 3-3. InfoParams Parameter Block

The application should check whether the driver is installed before making application calls. Check the driver software interrupt vector to see if the driver is installed (default is 0x60H).

- If that vector is zero, then the AppleTalk driver is not installed. If it is non-zero, then the driver might be installed.
- If the software interrupt vector is non-zero, examine the 16 bytes preceding the address to which this vector points. If those bytes are the ASCII characters AppleTalk, then the driver is installed. The byte immediately following the word "AppleTalk" is the minor version number. The next byte is the major version number, and the third byte is the driver type. For the AppleTalk driver, the type is a "1". Version 2.0 is stored as 200H. The four bytes following the version number are reserved by Apple, and are set to zero in Version 2.0.

ATInit (\$1)

The ATInit command initializes the driver software and announces the node on the network. Every application should make this call if a non-zero error code is returned from a ATGetNetInfo command.

The application may pass zero or suggest a node number. In this way, an application may choose a node number in the server range by suggesting a node number between 128 and 254. When the node is zero or when the suggested node number is already being used, the driver selects a node that is not in use. If the suggested node was in the server range, the node that the driver chooses dynamically will also fall within that range.

If the driver has already been initialized, the ATInit command will return the current node number. If there is an error, atd_status will have a negative number upon return. If there is no error, the actual node number will be returned in inf_nodeid.

Parameter usage and error returns for the ATInit command are as follows:

Parameter Usage

Input

-->	atd_command	ATInit
-->	atd_compfun	address of completion routine or zero
-->	inf_nodeid	suggested node number

Output

<--	atd_status	error code
<--	inf_nodeid	actual node ID

Error Returns

BAD_PARAMETER

Figure 3-4 illustrates the parameter block for the ATInit command.

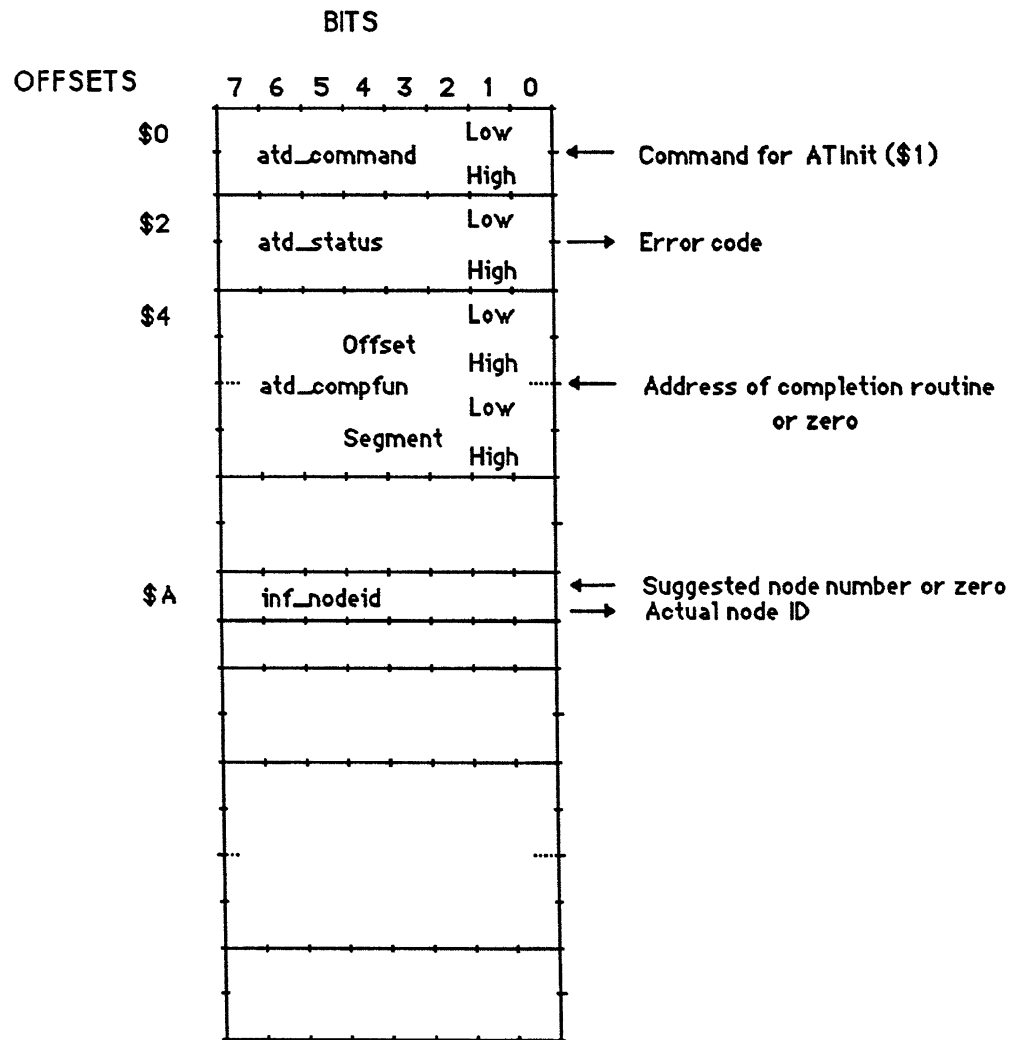


Figure 3-4. ATInit Parameter Block

ATKill (\$2)

The ATKill command effectively removes the node issuing the call from the network and then turns off the driver software. This call should never be used by an application, but should be made only by a special program specifically written to disconnect AppleTalk.

Parameter usage and error returns for the ATKill command are as follows:

Parameter Usage

Input

-->	atd_command	ATKill
-->	atd_compfun	address of completion routine or zero

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED

Figure 3-5 illustrates the parameter block for the ATKill command.

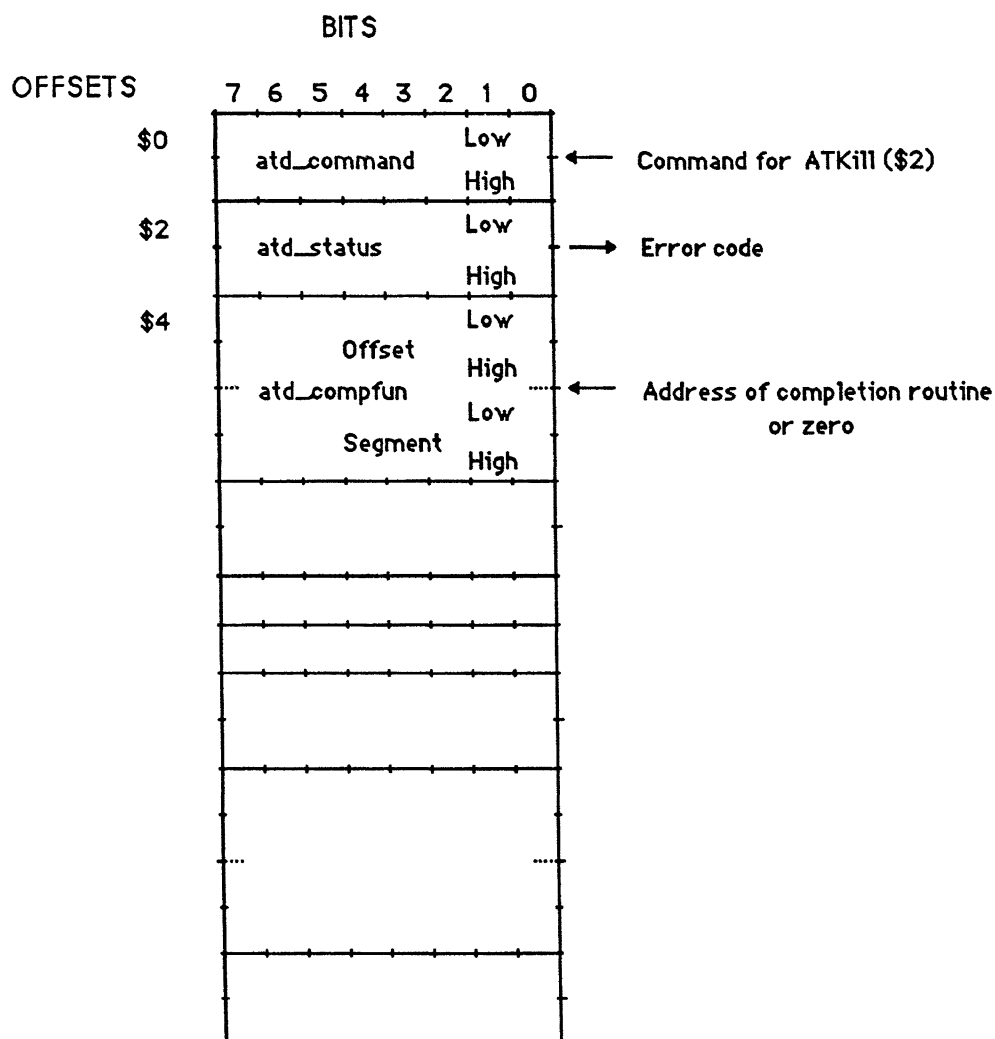


Figure 3-5. ATKill Parameter Block

ATGetNetInfo (\$3)

The application should use the ATGetNetInfo command whenever the application needs information about the network (usually at the start of a session). This command checks the installed state and indicates whether or not the node calling ATGetNetInfo has been initialized on the network.

If the driver is installed, the inf_config field will return a bit map giving configuration information for that driver. None of the other fields are valid (unless the driver is initialized, as described in this section). The bits in inf_config indicate which driver services and AppleTalk protocols are supported, with the different groups corresponding to the subsequent bit positions. All of the other bits are reserved for future extensions. Table 3-1 lists services and protocols offered by the driver to the application.

Table 3-1. Configuration Fields

Protocol/Service	Bit Number	Mask
Timer	0	1
LAP	1	2
DDP	2	4
NBP	3	8
ATP	4	16
ZIP	5	32
PAPWks	6	64
PAPSrvr	7	128
ASPWks	8	256
ASPSrvr	9	512
ADSP	10	1024
Special Services	15	32768

If the driver is initialized, the ATGetNetInfo call returns information regarding the network address of the calling node, the address of a bridge to another network (if one is known), and optional error and status information (driver dependent).

Parameter usage and error returns for the ATGetNetInfo command are as follows:

Parameter Usage

Input

--> atd_command	ATGetNetInfo
--> atd_status	set to -1 for installation check
--> atd_compfun	address of completion routine or zero
--> inf_buffptr	segment:offset of buffer area
--> inf_buffsize	size of buffer data

Output

<-- atd_status	error code
<-- inf_network	network number or zero
<-- inf_nodeid	node ID or zero
<-- inf_abridge	node of a Bridge or zero
<-- inf_config	Driver configuration mask

Error Returns

ATNOTINITIALIZED
BAD_PARAMETER

Figure 3-6 illustrates the parameter block for the ATGetNetInfo command.

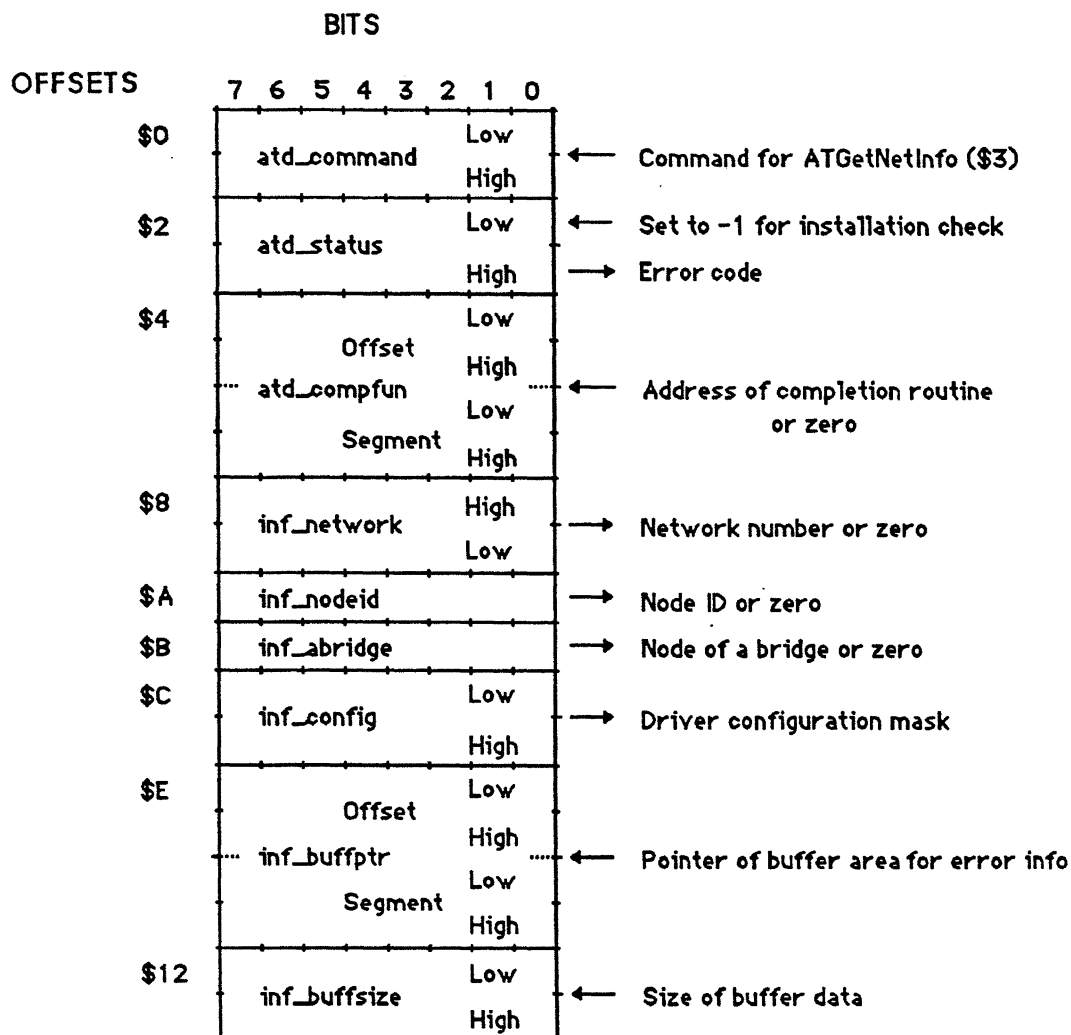


Figure 3-6. ATGetNetInfo Parameter Block

If the driver has been initialized (if atd_status equals NOERR), then inf_nodeid will contain the node number of that station, inf_network will return the node's network number (or 0 if it's not known), and inf_abridge will return the node of a bridge (or 0 if none are known). If the driver has not been initialized, atd_status equals ATNOTINITIALIZED. Use the ATInit command, described earlier in this chapter, to initialize the driver.

ATGetNetInfo also returns error and status information. If status information is desired, the application passes to the driver the address of a buffer and its size in `inf_buffptr` and `inf_buffsize`. If status information is not desired, the `inf_buffsize` field should be 0. If requested by the application, the driver returns status and error information in the passed buffer formatted as an `ATErrorStatus` structure. The last element indicates an arbitrary buffer where the driver can place hardware-specific or driver-specific information.

ATErrorStatus

The `ATErrorStatus` structure describes the general error and status information supplied by `ATGetNetInfo`.

```
typedef struct ATErrorStatus
{
    int    buflen;           /* number of valid bytes in buffer*/
    long   packets;         /* number of packets received */
    long   crcerrs;         /* number of CRC errors detected */
    long   overruns;        /* number of overruns detected */
    long   deferrals;        /* number of deferrals detected */
    long   collisions;      /* number of collisions detected */
    long   tossed;          /* number of packets tossed */
    long   mem_free;        /* number of Bytes of Free Memory */
    long   largest_mem;     /* Largest contiguous chunk of
                           memory available */
    long   int_bytes;       /* number of bytes used by driver */
    long   ext_bytes;       /* number of bytes used by external
                           aps */
}    ATErrorStatus;
```

Figure 3-7 illustrates the parameter block for the `ATErrorStatus` command.

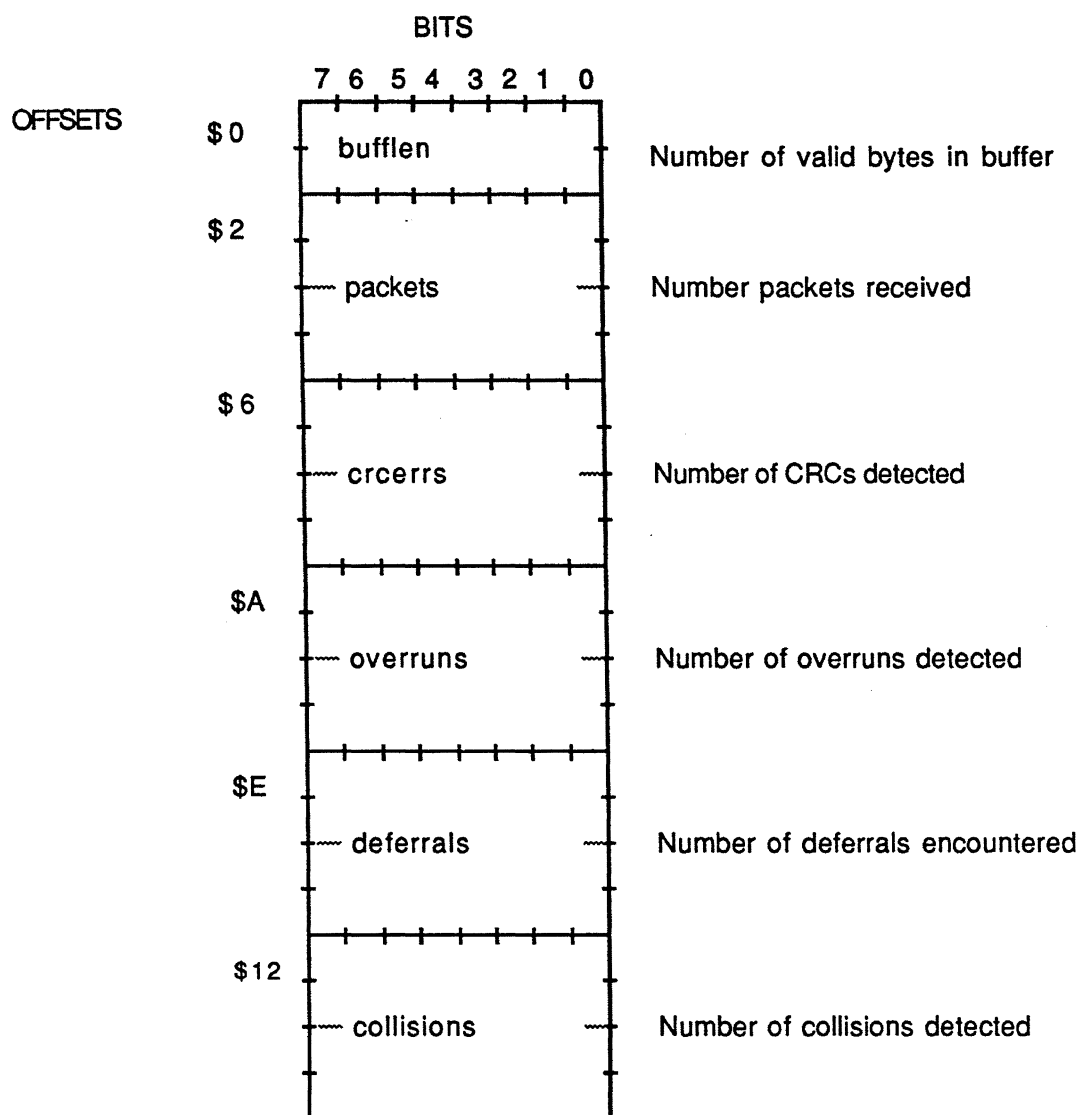


Figure 3-7. ATErrorStatus Parameter Block
Continued on next page

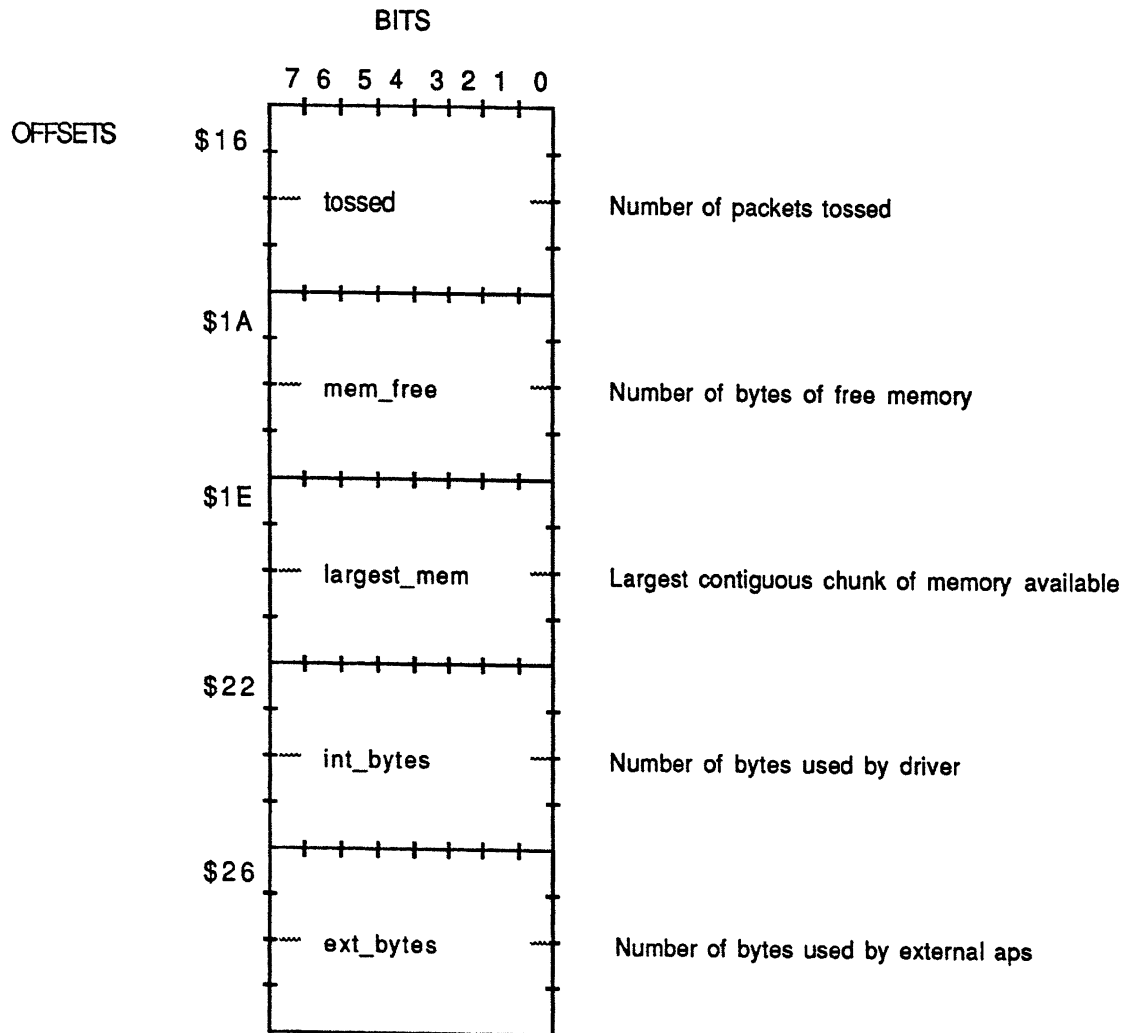


Figure 3-7. AErrorStatus Parameter Block Continued

Timer Commands

This group of timer commands provides convenient access to a timer mechanism on PCs. Four Timer-related commands are described in this section:

- ATGetClockTicks
- ATStartTimer
- ATResetTimer
- ATCancelTimer

These commands allow the programmer to access the PC's system clock safely from completion routines. ATGetClockTicks simply returns the number of clock ticks (once each 55 milliseconds) since the driver was loaded. The other timer commands allow the setting up of service (timer) routines to be called by the driver after a specified time (in units of 1/3 seconds) has elapsed. These calls also always return the number of ticks.

The following structure is used in the interface to the Timer commands:

TimerParams		
word	atd_command;	command code
word	atd_status;	status word/error return
dword	atd_compfun;	completion (timer) routine or zero
dword	tmr_ticks;	timer ticks (since init.)
word	tmr_time;	number of 1/3 seconds to wait
dword	tmr_params;	address of a TimerParams structure

Figure 3-8 illustrates the general parameter block used in the interface to the Timer commands.

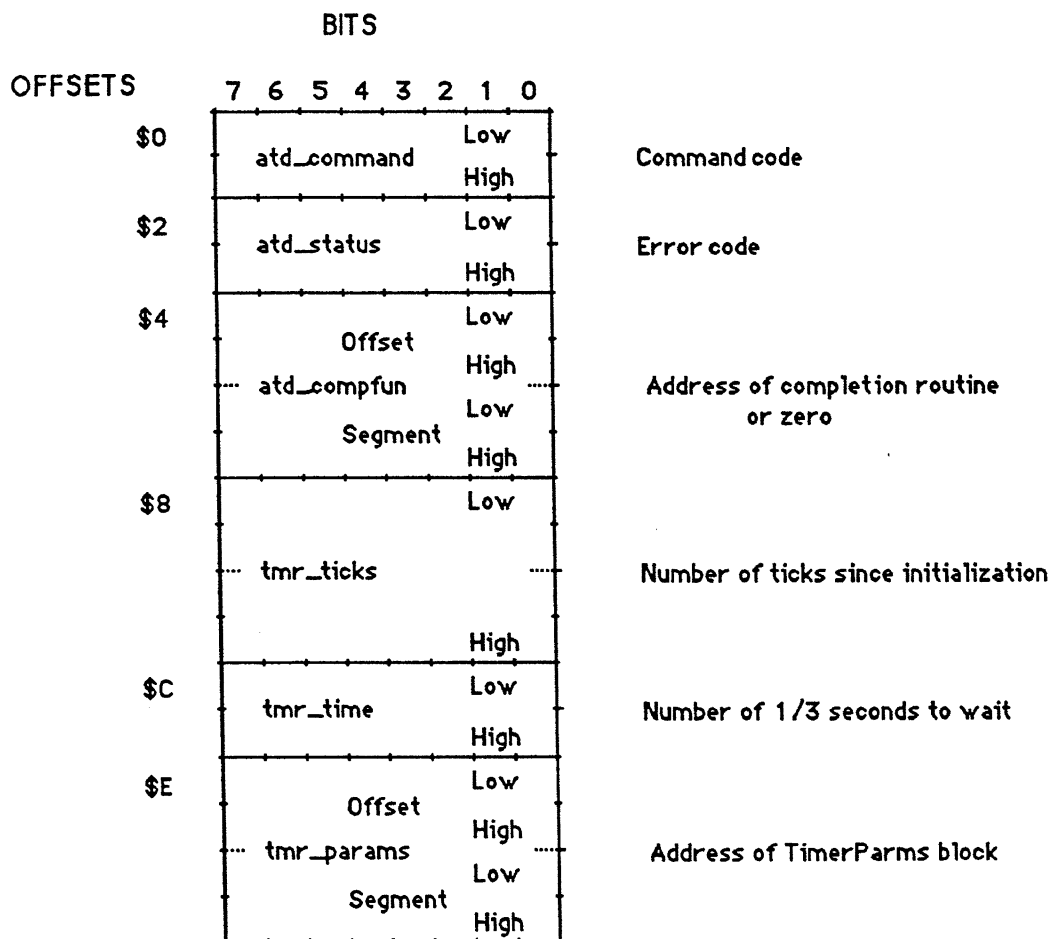


Figure 3-8. General Parameter Block for Timer Commands

ATGetClockTicks (\$4)

The ATGetClockTicks command returns in the `tmr_ticks` field the number of clock ticks since the driver was initialized. Clock ticks are approximately 55 milliseconds (1/18.2 second).

Parameter usage and error returns for the ATGetClockTicks command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	ATGetClockTicks
-->	<code>atd_compfun</code>	address of completion (timer) routine or 0

Output

<--	<code>atd_status</code>	error code
<--	<code>tmr_ticks</code>	number of ticks

Error Returns

NOERR
ATNOTINITIALIZED

Figure 3-9 illustrates the parameter block for the ATGetClockTicks command.

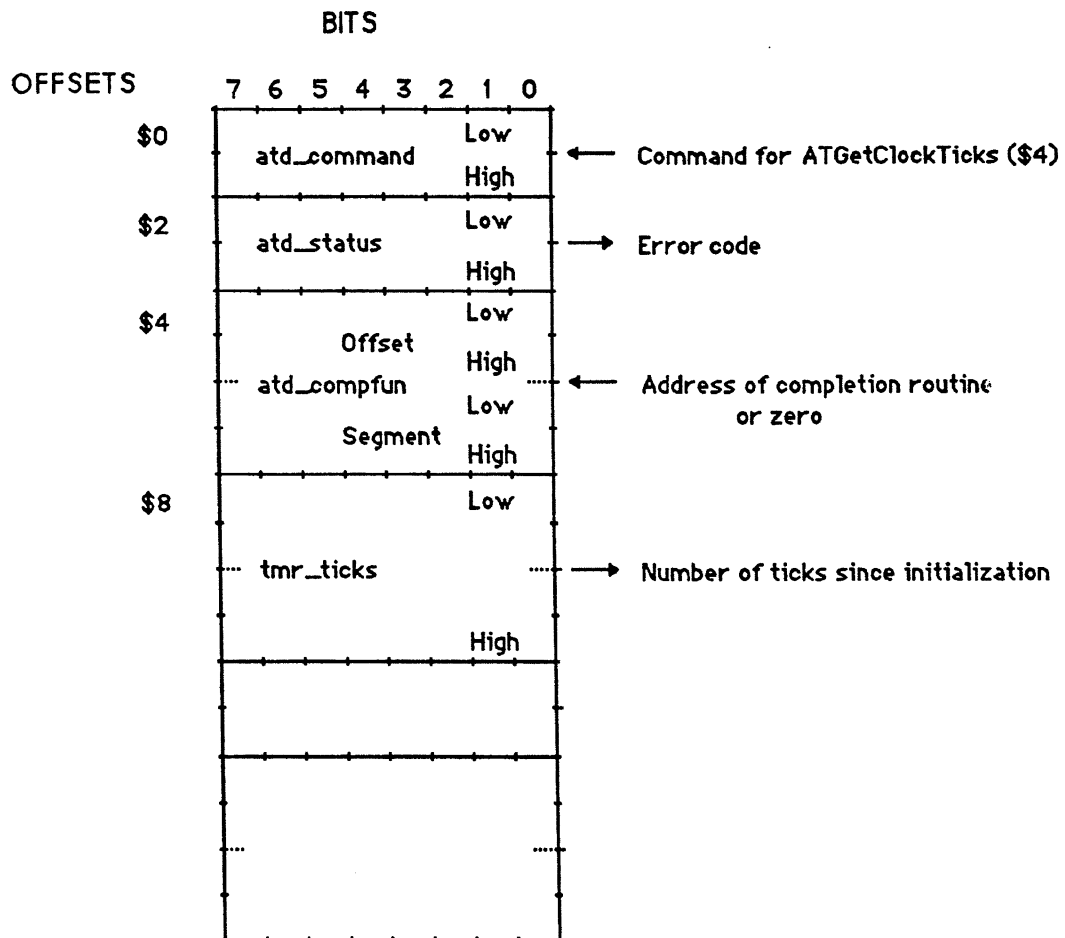


Figure 3-9. ATGetClockTicks Parameter Block

ATStartTimer (\$5)

The ATStartTimer command lets the caller set up a timer-activated routine to be called when the specified time (in units of 1/3 seconds) has elapsed.

To use this service, the caller must set the `tmr_time` field to the appropriate number. If the caller specifies a completion (timer) routine, the routine will be called as described in "Completion Routines" in Chapter 2. The driver returns with the `tmr_ticks` field updated. If the caller does not specify a timer routine, the caller may monitor the status word, which will be set to zero when the specified time has elapsed.

Parameter usage and error returns for the ATStartTimer command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	ATStartTimer
-->	<code>atd_compfun</code>	address of completion (timer) routine or 0
-->	<code>tmr_time</code>	number of 1/3 seconds to wait

Output

<--	<code>atd_status</code>	error code
<--	<code>tmr_ticks</code>	number of ticks

Error Returns

NOERR
ATNOTINITIALIZED

Figure 3-10 illustrates the parameter block for the ATStartTimer command.

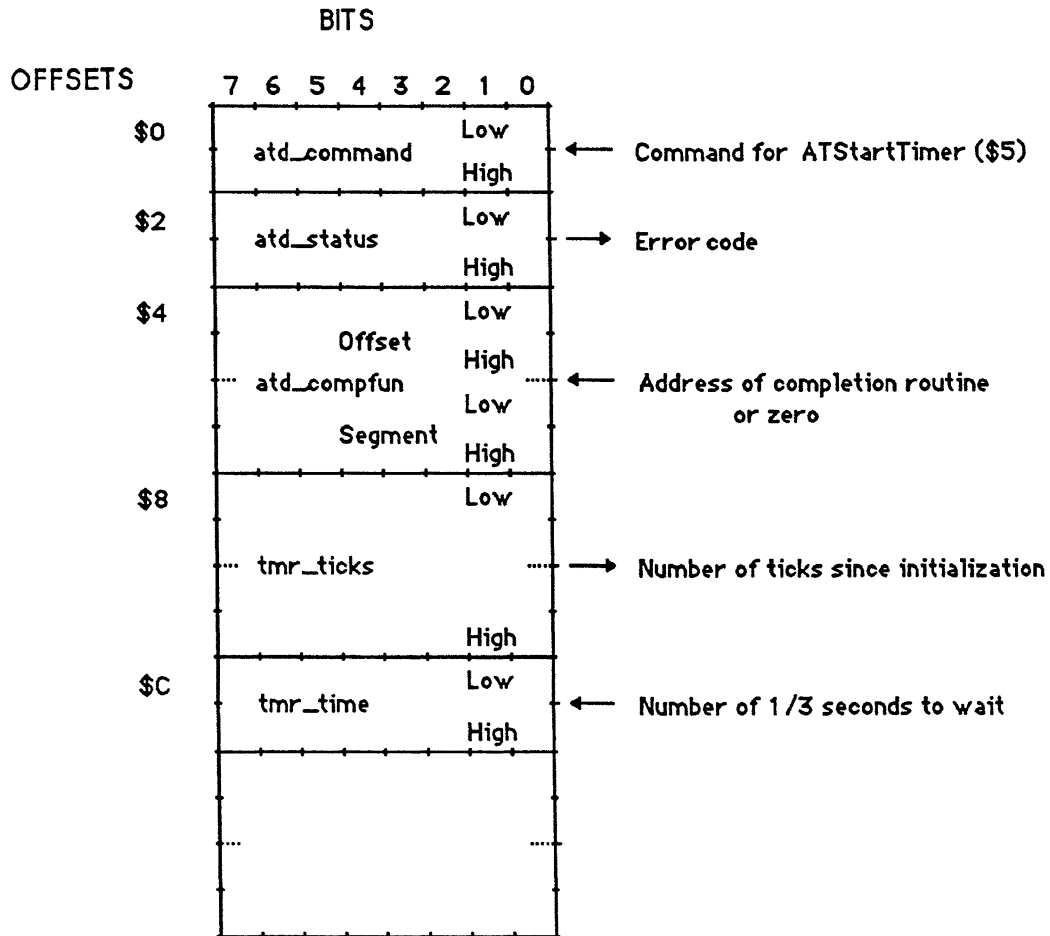


Figure 3-10. ATStartTimer Parameter Block

ATResetTimer (\$6)

The ATRestTimer command lets the caller reset the number of seconds associated with a timer routine already set up with ATStartTimer.

Making this call will use the tmr_time field to update the associated internal timer field of the TimerParams structure that the tmr_params field points to. If the caller uses this call after a timer has expired and the timer routine has been called, the TMR_NOT_FOUND error occurs.

Parameter usage and error returns for the ATRestTimer command are as follows:

Parameter Usage

Input

-->	atd_command	ATResetTimer
-->	atd_compfun	address of completion (timer) routine or 0
-->	tmr_time	number of 1/3 seconds to wait
-->	tmr_params	segment:offset of TimerParams block of timer being reset

Output

<--	atd_status	error code
<--	tmr_ticks	number of ticks

Error Returns

NOERR
ATNOTINITIALIZED
TMR_NOTFOUND

Figure 3-11 illustrates the parameter block for the ATRestTimer command.

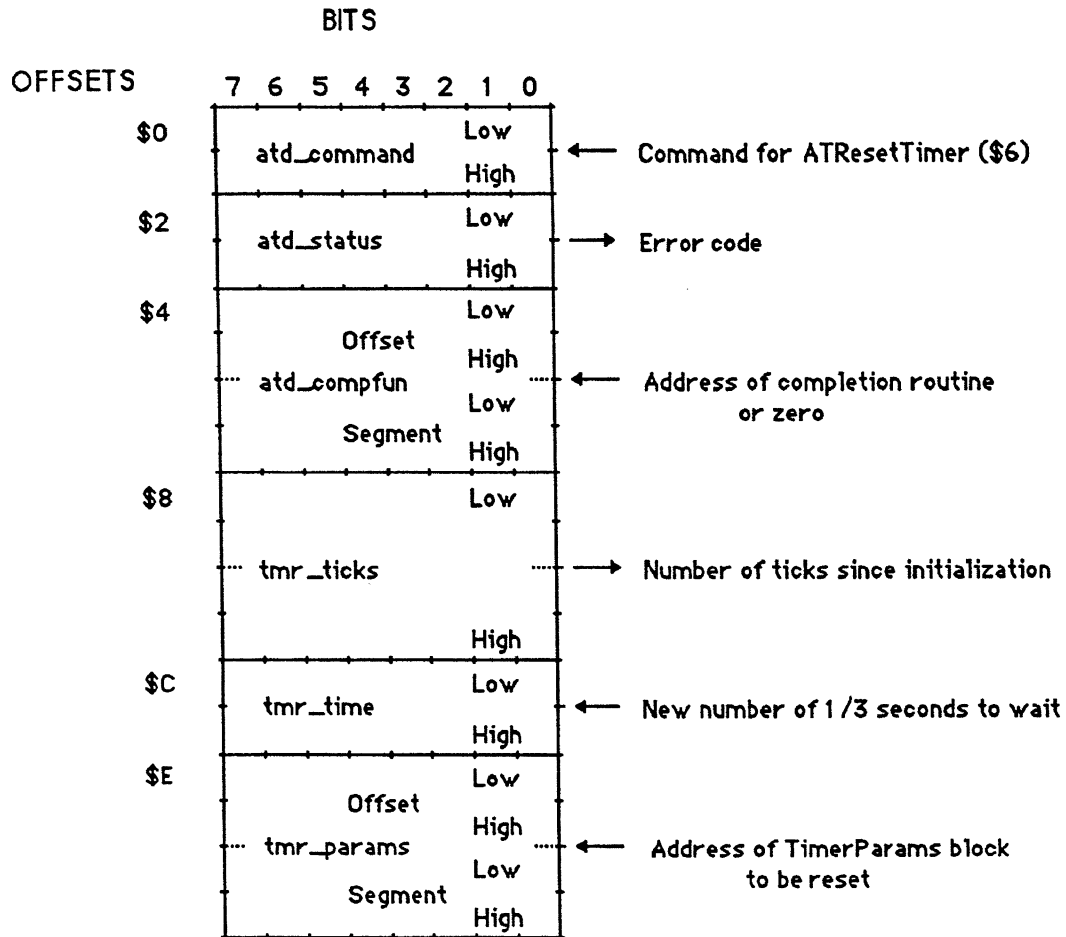


Figure 3-11. ATResetTimer Parameter Block

ATCancelTimer (\$7)

The ATCancelTimer command lets the caller cancel a timer already set up by a previous ATStartTimer. The tmr_params field should point to the TimerParams structure of the timer being cancelled. If the caller specifies a timer routine, the routine is called with the value TMR_CANCELLED in the atd_status word of the parameter structure.

Parameter usage and error returns for the ATCancelTimer command are as follows:

Parameter Usage

Input

--> atd_command	ATCancelTimer
--> atd_compfun	address of completion (timer) routine or 0
--> tmr_params	segment:offset of TimerParams block of timer being cancelled

Output

<-- atd_status	error code
<-- tmr_ticks	number of ticks

Error Returns

NOERR
ATNOTINITIALIZED
TMR_NOTFOUND

Figure 3-12 illustrates the parameter block for the ATCancelTimer command.

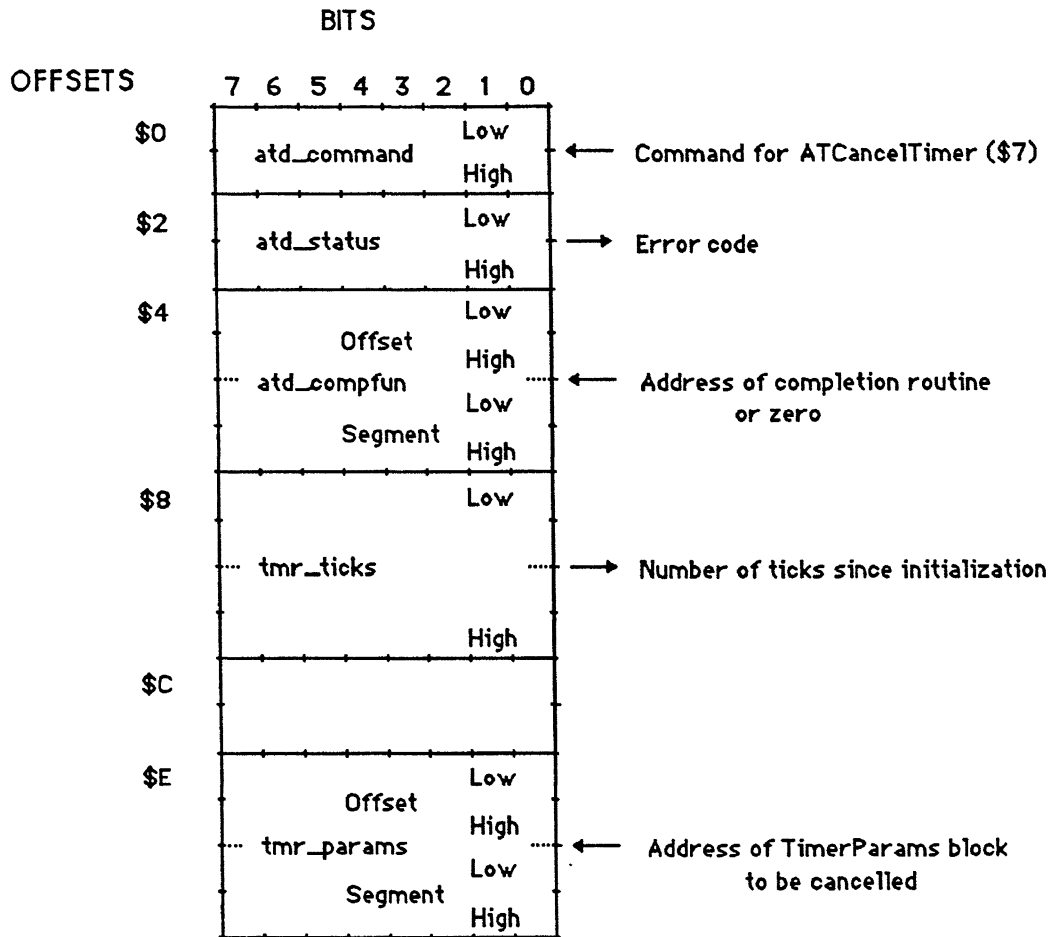


Figure 3-12. ATCancelTimer Parameter Block

Chapter 4

Link Access Protocol (LAP) Commands

Introduction

The Link Access Protocol (LAP) commands provide a low-level mechanism for data transfer within one network. LAP commands include

- LAPInstall
- LAPRemove
- LAPWrite
- LAPRead
- LAPCancel

In all cases, the LAP calls will return zero (NOERR) if the call is completed successfully, or a negative number if there is an error.

Table 4-1 lists the valid numerical quantities used in LAP commands.

Table 4-1. Valid LAP Type Field Values

Value	Description
\$00	Invalid LAP protocol type value (not to be used)
\$01 through \$7F	Valid LAP protocol type values for use in LAP client packets
\$01 through \$0F	Reserved for Apple Computer use only
\$01	DDP short header packet
\$02	DDP long header packet
\$0F	Experimental LAP packet (used by Apple only)
\$80 through \$FF	Reserved for LAP control frames
\$81	LAP ENQ packet
\$82	LAP ACK packet
\$84	LAP RTS packet
\$85	LAP CTS packet

Figure 4-1 illustrates the general LAP parameter block.

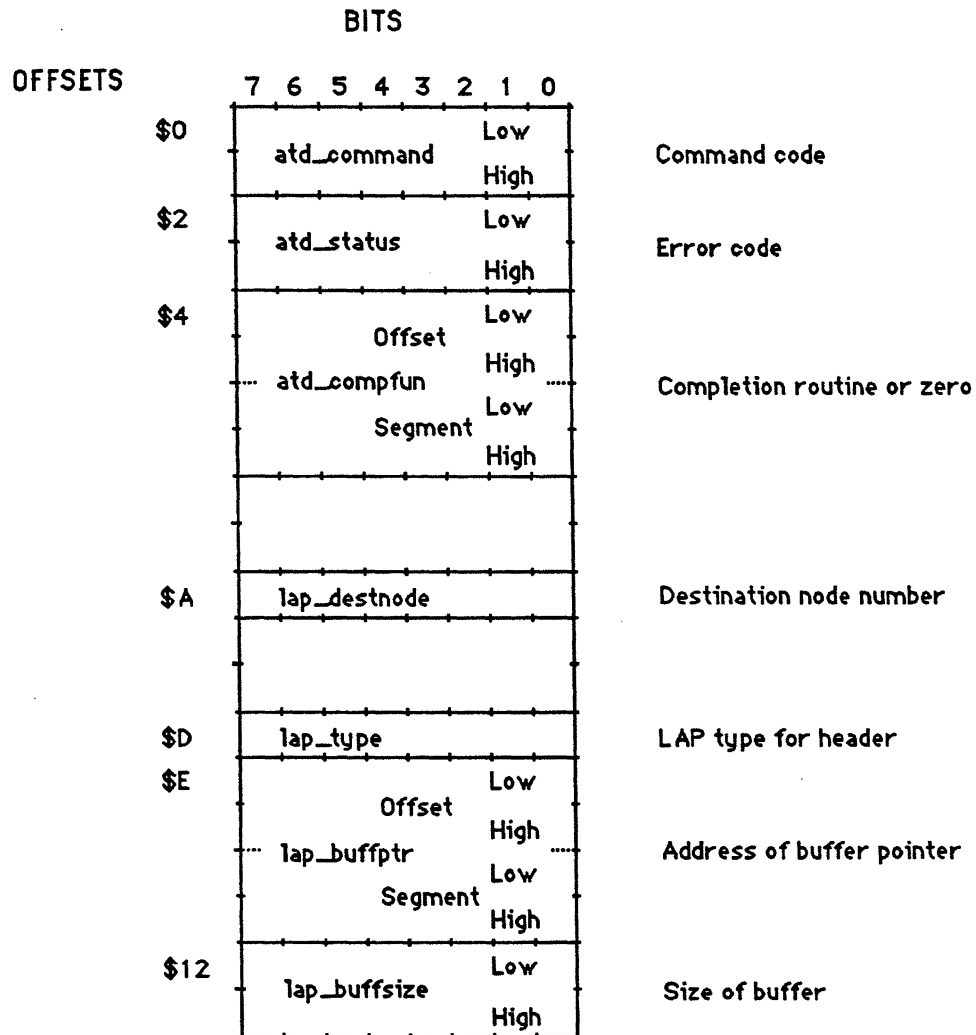


Figure 4-1. General Parameter Block for Link Access Protocol (LAP) Commands

LAPInstall (\$10)

The LAPInstall command installs a handler to take care of incoming packets of the LAP type specified in the lap_type field. Refer to Chapter 2 for a description of first-half and second-half handlers.

If the lap_buffptr field is not zero, the driver assumes that field holds the address of a protocol handler. After this command completes, the driver sends to the handler any packets with the specified type. If the lap_buffptr field is zero, the driver installs a default handler for that LAP type. Incoming packets will then be available through the LAPRead command, described later in this chapter.

As mentioned in "Calling Conventions" in Chapter 2, the handler is called with some restrictions. The user-installed protocol handler will *not* be called if there were any errors in the reception of the packet (such as CRC, overruns, time-outs, and so on). In addition, the first-half of the handler is called with the following inputs:

AH	LAP type
CX	Length of packet data
DS:BX	Segment:offset of packet header

The length passed to the handler in CX is the length of the packet data without the LAP header. The header pointed to by DS:BX is the three-byte LAP header. The first-half handler must decide to keep the packet or discard the data. If the packet is to be discarded, the first-half handler should return with CX equal to zero, in which case the second-half handler will not be called. If the data is to be retained, the first-half handler must return with the registers holding the following arguments:

CX	Size of buffer for data
DS:BX	Segment:offset of buffer area for packet
ES:DX	Segment:offset of second-half handler

The second-half handler is then called (through a FAR CALL) with the registers set as following:

CX	Number of bytes of data put in buffer
DS:BX	Segment:offset of buffer area for packet

The second-half of the handler may enable interrupts and should exit with a FAR RETURN.

Parameter usage and error returns for LAPInstall are as follows:

Parameter Usage

Input

-->	atd_command	LAPInstall
-->	atd_compfun	address of completion routine or 0
-->	lap_type	LAP type for handler
-->	lap_buff_ptr	segment:offset of protocol handler

Output

<--	atd_status	error code
-----	------------	------------

AppleTalk PC Card and Driver Preliminary Note

Error Returns

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER
 TABFULLERR
 LAP_TYPERR

Figure 4-2 illustrates the parameter block for the LAPInstall command.

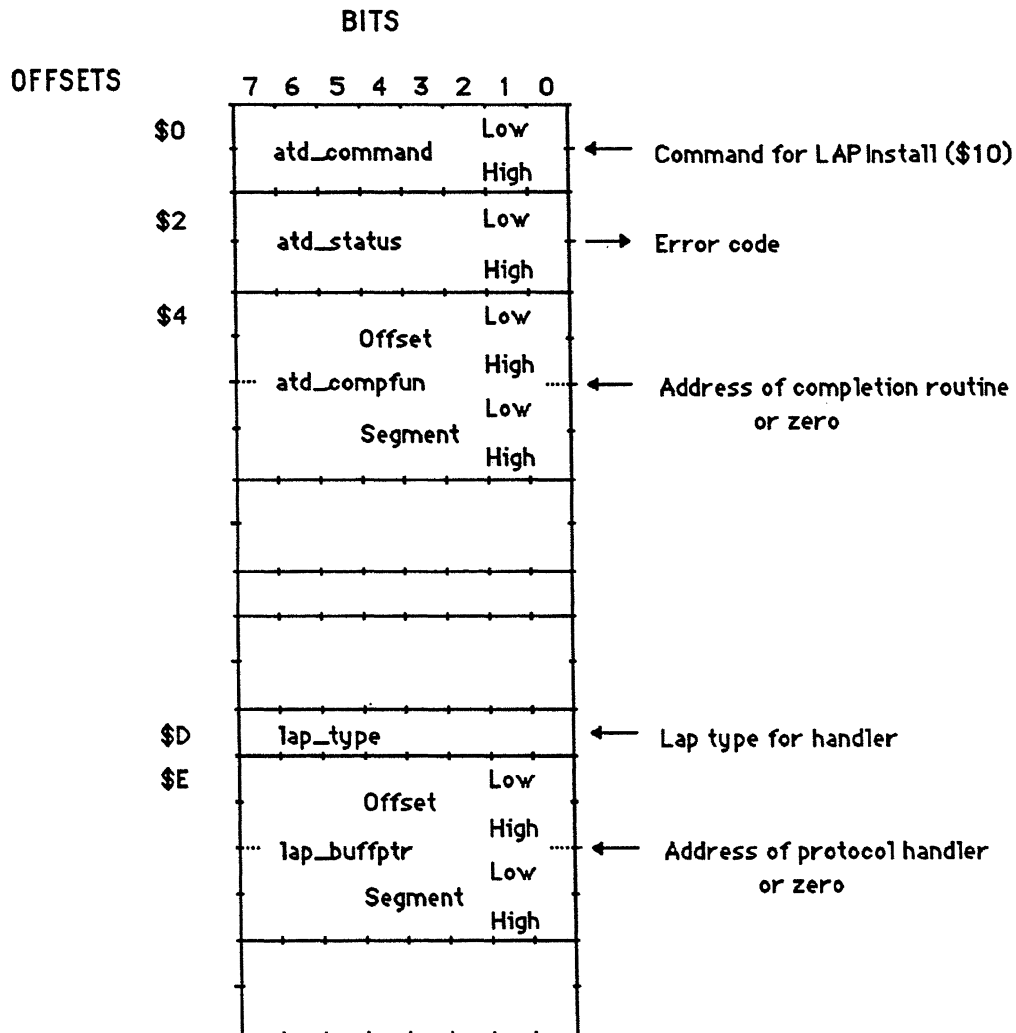


Figure 4-2. LAPInstall Parameter Block

LAPRemove (\$11)

The LAPRemove command removes the handler for the specified LAP type from the nodes protocol table.

Note: LAPRemove cannot remove protocol types 1 or 2.

Parameter usage and error returns for LAPRemove are as follows:

Parameter Usage

Input

-->	atd_command	LAPRemove
-->	atd_compfun	address of completion routine or 0
-->	lap_type	LAP type of handler to remove

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
LAP_NOTFND
LAP_TYPERR

Figure 4-3 illustrates the parameter block for the LAPRemove command.

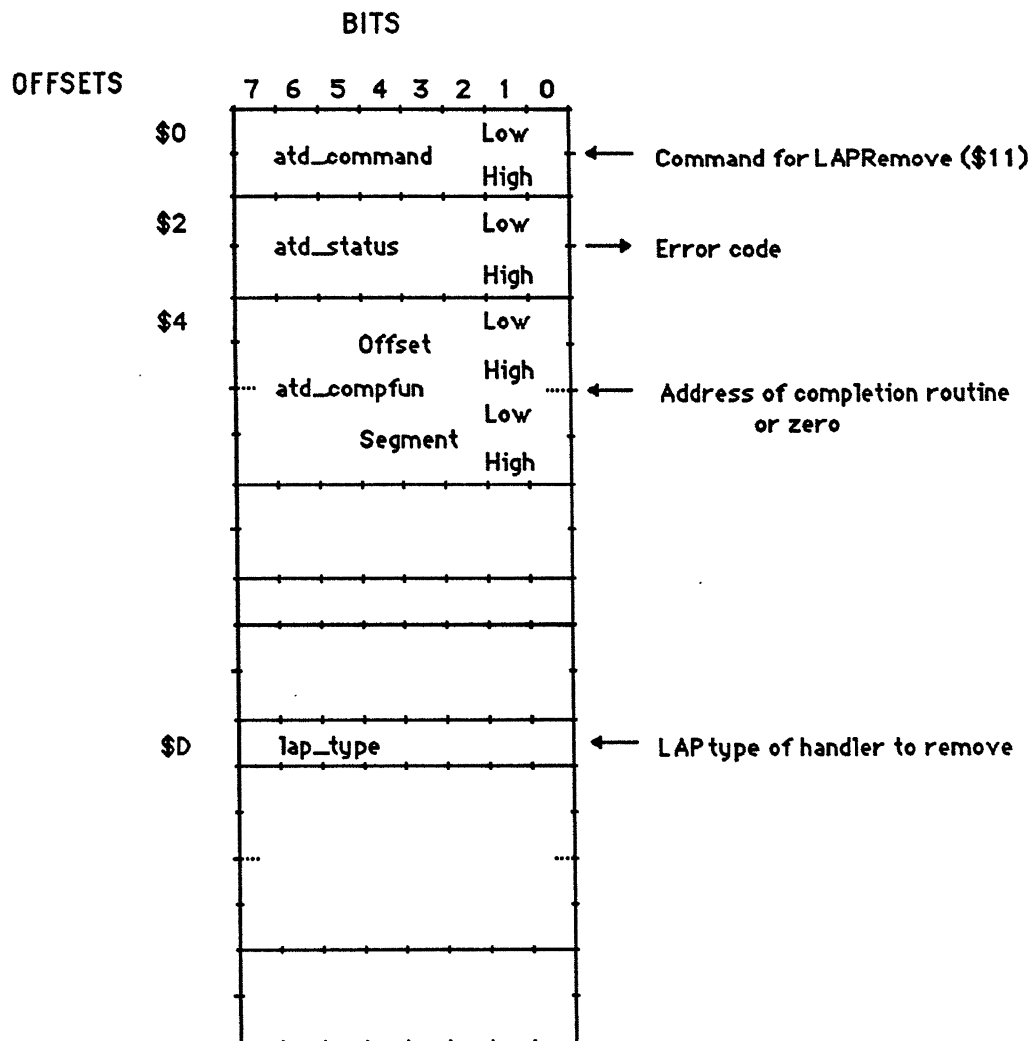


Figure 4-3. LAPRemove Parameter Block

LAPWrite (\$12)

The LAPWrite command sends any data to the specified node.

The limit on the length of the data to be sent is 600 bytes. The first two bytes of the LAP data portion of the packet must hold the length of the LAP data (consistent with DDP header formats). The lap_buffsize field should include the two bytes used by that field in the count. In addition, the length field (first word) in the buffer should be "high byte first" (68000 ordering), although the length in the parameter structure should be in standard 8086 byte ordering. The lap_type field will be placed in LAP header.

The following structure is used in the interface to LAPWrite:

LAPParams

word atd_command;	command code
word atd_status;	status word/error return
dword atd_compfun;	completion routine or zero
word lap_fill1;	filler word
byte lap_destnode;	destination node number
word lap_fill2;	filler word
byte lap_type;	LAP type for header
dword lap_buffptr;	segment:offset of buffer area or protocol handler (LapInstall)
word lap_buffsize;	size of buffer area

Parameter usage and error returns for LAPWrite are as follows:

Parameter Usage

Input

--> atd_command	LAPWrite
--> atd_compfun	address of completion routine or 0
--> lap_destnode	destination node
--> lap_type	LAP type for header
--> lap_buffptr	segment:offset of packet data
--> lap_buffsize	size of packet data

Output

<-- atd_status	error code
----------------	------------

Error Returns

NOERR
 ATNOTINITIALIZED
 LAP_LENERR
 STACKERROR
 NO_MEM_ERROR
 BAD_PARAMETER
 MAXCOLISERR
 MAXDEFERERR
 LAP_TYPERR

Figure 4-4 illustrates the parameter block for the LAPWrite command.

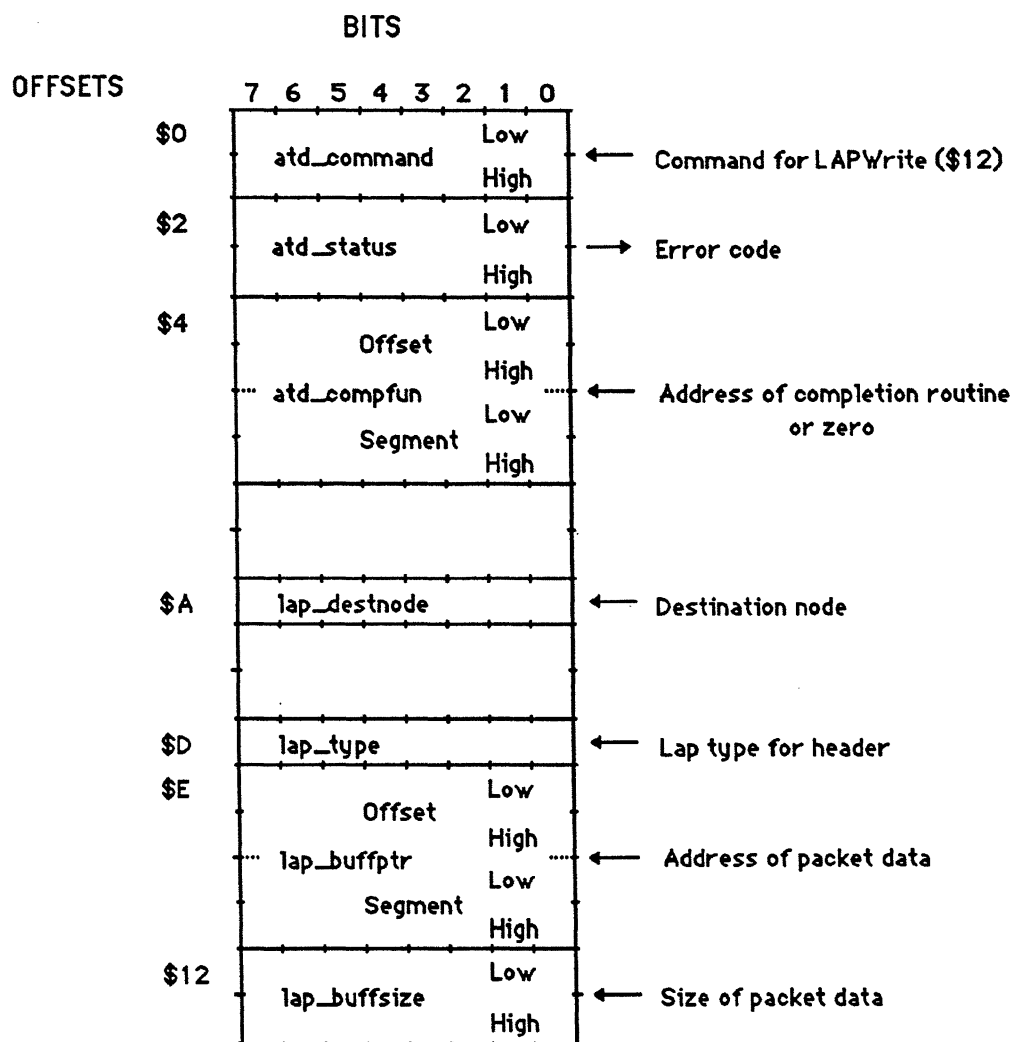


Figure 4-4. LAPWrite Parameter Block

LAPRead (\$13)

The LAPRead command receives any data sent to the specified node with the lap type specified by the lap_type field. This command uses the default protocol handler to read the packet, and can be used only if no user protocol handler is installed. If the lap_buffptr field is set by the caller, the entire contents of the packet will be placed at the passed buffer. The lap_buffsize field will be updated to reflect the number of bytes placed in the buffer.

There is one special case of this command. If this call is made with the lap_buffptr field set to zero, the driver will read the incoming packet into an internal buffer. The driver will then call the completion routine (which *must* be specified to get the data), with the lap_buffptr field set to the *segment:offset* of this internal buffer. The completion routine will then be able to read the packet from this buffer and save the contents locally if it chooses to do so.

Note: The case just described is an exception to the calling conventions for completion routines. In this case, the routine is called with interrupts disabled, and the completion routine should *not* enable interrupts until it has safely copied the packet contents to a local buffer.

Parameter usage and error returns for the LAPRead command are as follows:

Parameter Usage

Input

-->	atd_command	LapRead
-->	atd_compfun	address of completion routine or 0
-->	lap_type	lap type for incoming
-->	lap_buffptr	segment:offset of buffer or 0
-->	lap_buffsize	size of buffer

Output

<--	atd_status	error code
<--	lap_destnode	node of sender
<--	lap_buffsize	size of packet in buffer
<--	lap_buffptr	buffer address (if 0 on input)

Error Returns

```
NOERR
ATNOTINITIALIZED
LAP_LENERR
LAP_NOTFND
BAD_PARAMETER
LAP_CANCELLED
LAP_TYPERR
```

Figure 4-5 illustrates the parameter block for the LAPRead command.

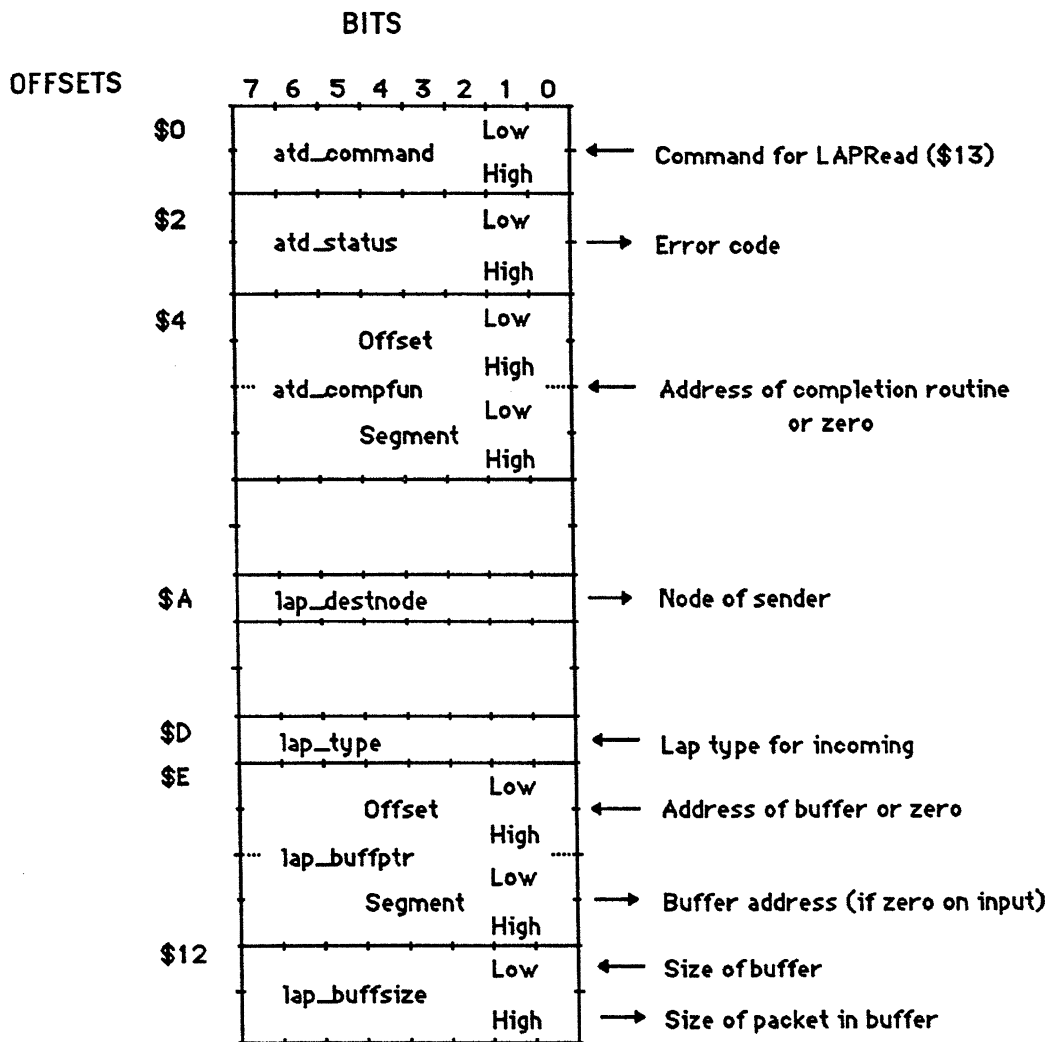


Figure 4-5. LAPRead Parameter Block

LAPCancel (\$14)

The LAPCancel command cancels any outstanding LAP command. The parameter structure *segment:offset* passed in the *lap_buffptr* field must be the same one as that passed in with the command being cancelled.

Parameter usage and error returns for the LAPCancel command are as follows:

Parameter Usage

Input

-->	<i>atd_command</i>	LAPCancel
-->	<i>atd_compfun</i>	address of completion routine or 0
-->	<i>lap_buffptr</i>	params for command being cancelled

Output

<--	<i>atd_status</i>	error code
-----	-------------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER

Figure 4-6 illustrates the parameter block for the LAPCancel command.

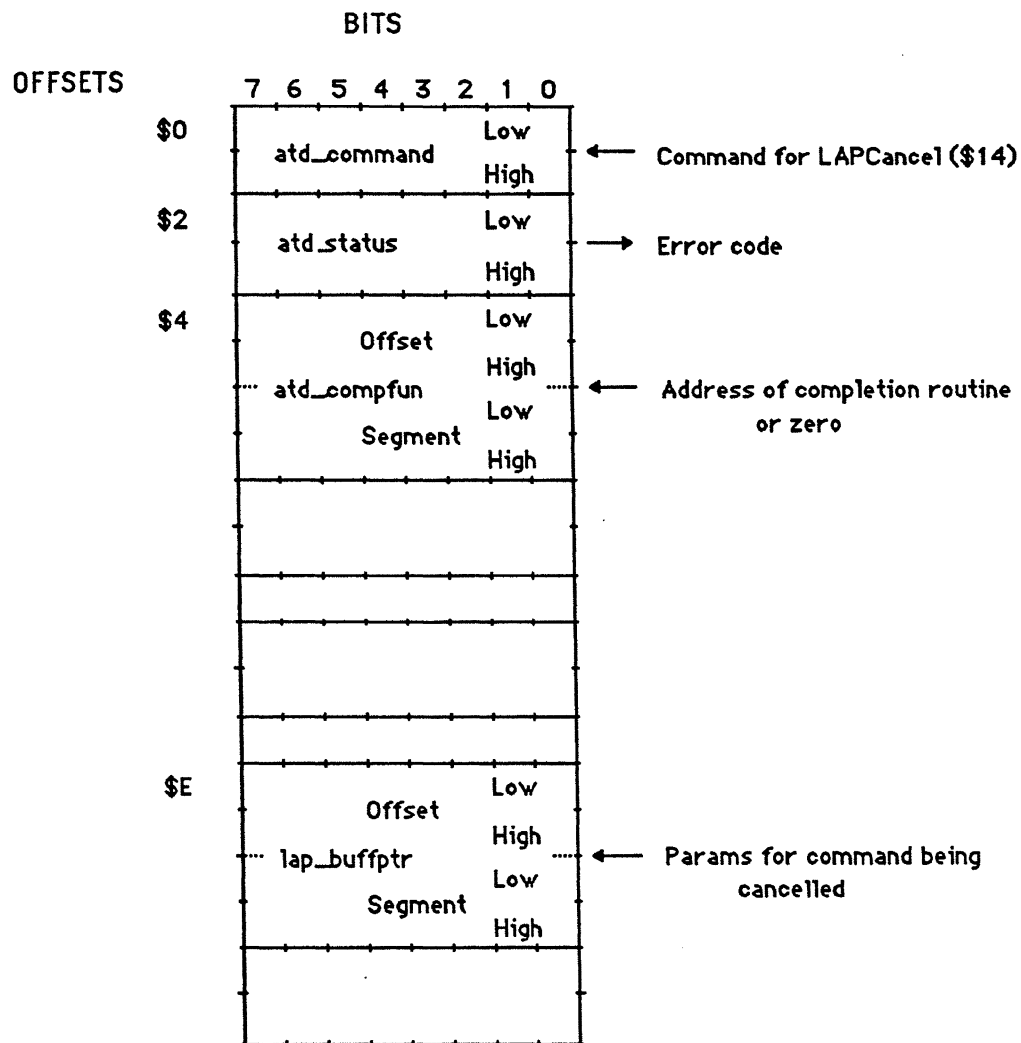


Figure 4-6. LAPCancel Parameter Block

Chapter 5

Datagram Delivery Protocol (DDP) Commands

Introduction

The Datagram Delivery Protocol (DDP) provides a low-level mechanism for transporting data between sockets on networks or within a single network. DDP commands include

- DDPOpenSkt
- DDPCloseSkt
- DDPWrite
- DDPRead
- DDPCancel

In all cases, DDP calls will return zero (0 = NOERR) to indicate no errors if the call is completed successfully, or a negative error code if there is an error.

The driver does not impose a “fixed” limit on the number of sockets that may be open at a time. Memory is allocated from the driver pool until exhausted. The memory available may depend upon the initial memory size, as well as other nondependent operations that may draw upon the memory pool.

Table 5-1 lists the valid numerical quantities used in for DDP commands.

Table 5-1. Valid DDP Type Field Values

Value	Description
\$00	Invalid DDP protocol type value (not to be used)
\$01 through \$FF	Valid DDP protocol type values for use in DDP client packets
\$01 through \$0F	Reserved for use only by Apple Computer, Inc.
\$01	RTMP response or data packet
\$02	NBP packet
\$03	ATP packet
\$04	EP (Echo Protocol) packet
\$05	RTMP request packet
\$06	ZIP packet
\$07	ASP packet

Table 5-2 lists the socket numbers available for DDP commands.

Table 5-2. Valid Socket Numbers

Value	Description
\$00	Invalid (do not use)
\$FF	Invalid (do not use)
\$01 through \$FE	Valid DDP sockets
\$01 through \$7F	Statically assigned sockets
\$01 through \$3F	Reserved for use only by Apple Computer, Inc.
\$40 through \$7F	Experimental use only (<i>not</i> to be used in released products)
\$01	RTMP socket
\$02	Names Information socket
\$04	Echoer socket
\$06	Zone Information socket

Figure 5-1 illustrates the general DDP parameter block.

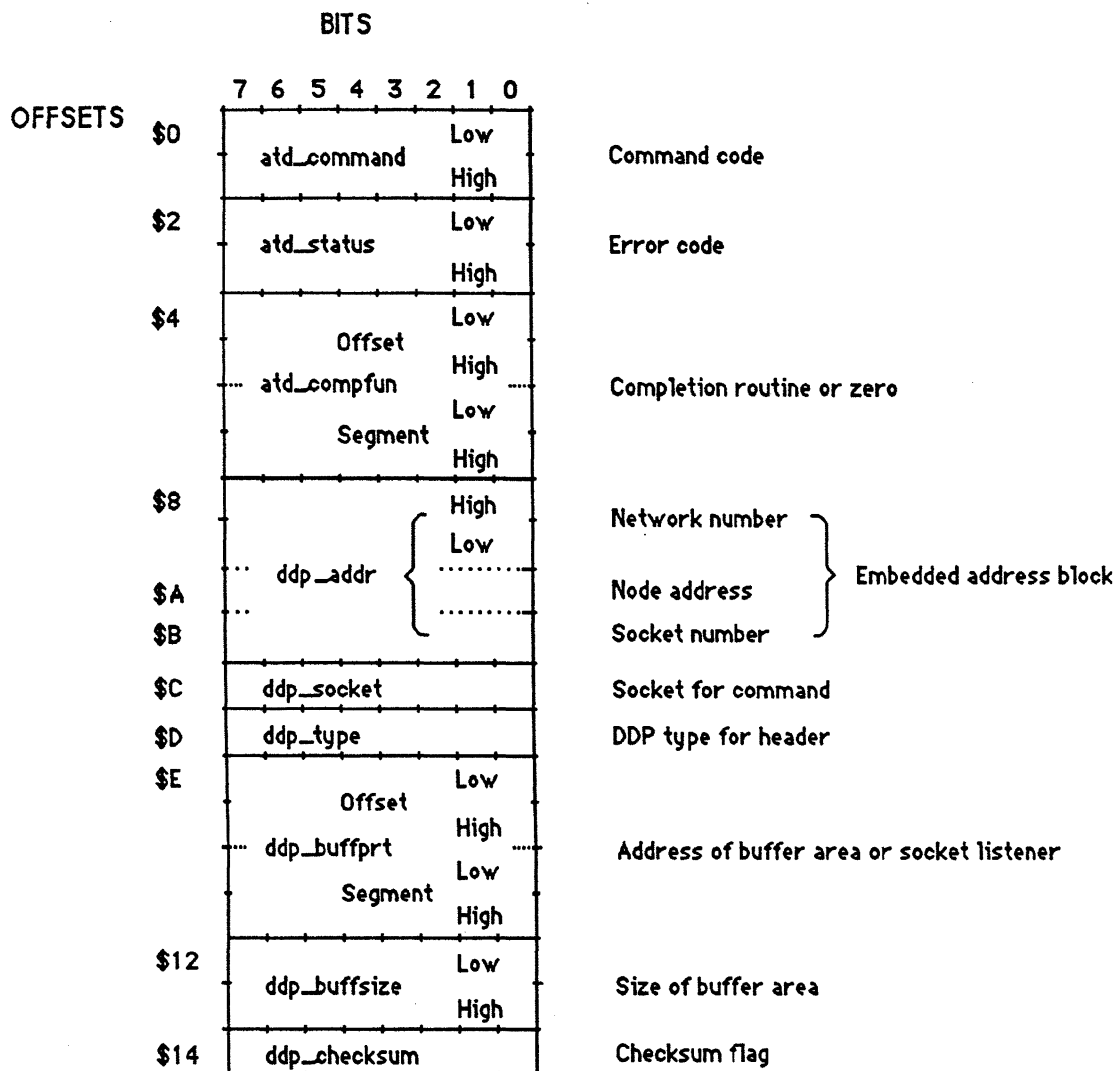


Figure 5-1. General Parameter Block for Datagram Delivery Protocol (DDP) Commands

Description of the DDP commands are provided in the remainder of this chapter, along with examples of parameter blocks and a list of error returns for each command.

DDPOpenSkt (\$20)

The DDPOpenSkt call installs a listener for the passed socket. If the ddp_socket field is zero, the driver assigns a socket number dynamically. The socket number is passed back in the ddp_socket field. If the ddp_buffptr field is not zero, it is assumed to hold the address of a socket listener.

After this command, any packets directed to the new socket will be sent to the listener. If the ddp_buffptr field is zero, a default listener for that socket will be installed. Incoming packets will then be available through the DDPRead command, described later in this chapter.

As mentioned in "Calling Conventions" in Chapter 2, the listener is called with some restrictions. In addition, the first-half of the listener is called with the following inputs:

AH	Socket
CX	Length of data
DS:BX	Segment:offset of packet (LAP & DDP) header

The first-half of the listener must decide to keep the packet or discard the data according to these guidelines:

- If the packet is to be discarded, the first-half listener should return with CX equal to zero, in which case the second-half listener will not be called.
- If the data is to be retained, the first-half listener must return with the registers holding the following arguments:

CX	Size of buffer for data
DS:BX	Segment:offset of buffer area for packet
ES:DX	Segment:offset of second-half listener

The second-half of the listener is then called (through a FAR CALL), with the registers set as follows:

CX	Number of bytes of data put in buffer
DS:BX	Segment:offset of buffer area for packet

The second-half of the listener may enable interrupts and should exit with a FAR RETURN.

Note: If the checksum field of the DDP header is set, the user's socket listener is responsible for checking that the Datagram has been received correctly by verifying the checksums as described in *Inside AppleTalk*.

Parameter usage and error returns for the DDPOpenSkt command are listed on the following page.

AppleTalk PC Card and Driver Preliminary Note

Parameter Usage

Input

```
--> atd_command      DDPOpenSocket
--> atd_compfun      address of completion routine or 0
--> ddp_socket       suggested socket number
--> ddp_buffptr      address of listener or 0
```

Output

```
<-- atd_status       error code
<-- ddp_socket       actual socket number
```

Error Returns

```
NOERR
ATNOTINITIALIZED
BAD_PARAMETER
```

Figure 5-2 illustrates the parameter block for the DDPOpenSkt command.

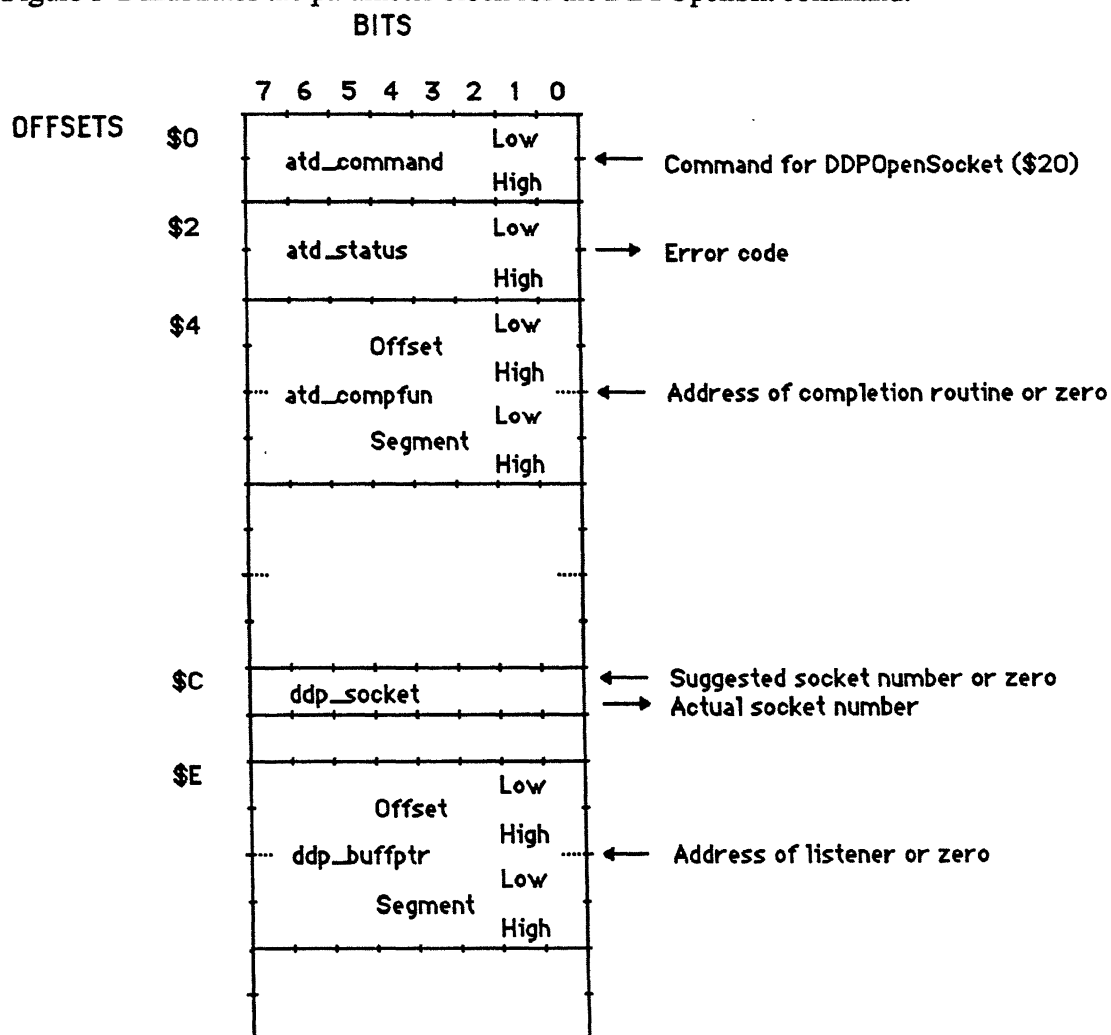


Figure 5-2. DDPOpenSkt Parameter Block

DDPCloseSkt (\$21)

The DDPCloseSkt call removes the listener of the specified socket from the socket table.

DDPCloseSocket cannot close socket 1 (RTMP's socket) or socket 2 (NBP's socket). It is not advisable for the application to close a socket with DDPCloseSocket that was opened with ATPOpenSocket. If this is done, outstanding ATP transactions on the socket will not be cancelled; if the application ever reopens the socket, these outstanding transactions will become reactivates.

Parameter usage and error returns for the DDPCloseSkt command are as follows:

Parameter Usage

Input

-->	atd_command	DDPCloseSocket
-->	atd_compfun	address of completion routine or 0
-->	ddp_socket	socket to be closed

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
DDP_SKTERR

Figure 5-3 illustrates the parameter block for the DDPCloseSkt command.

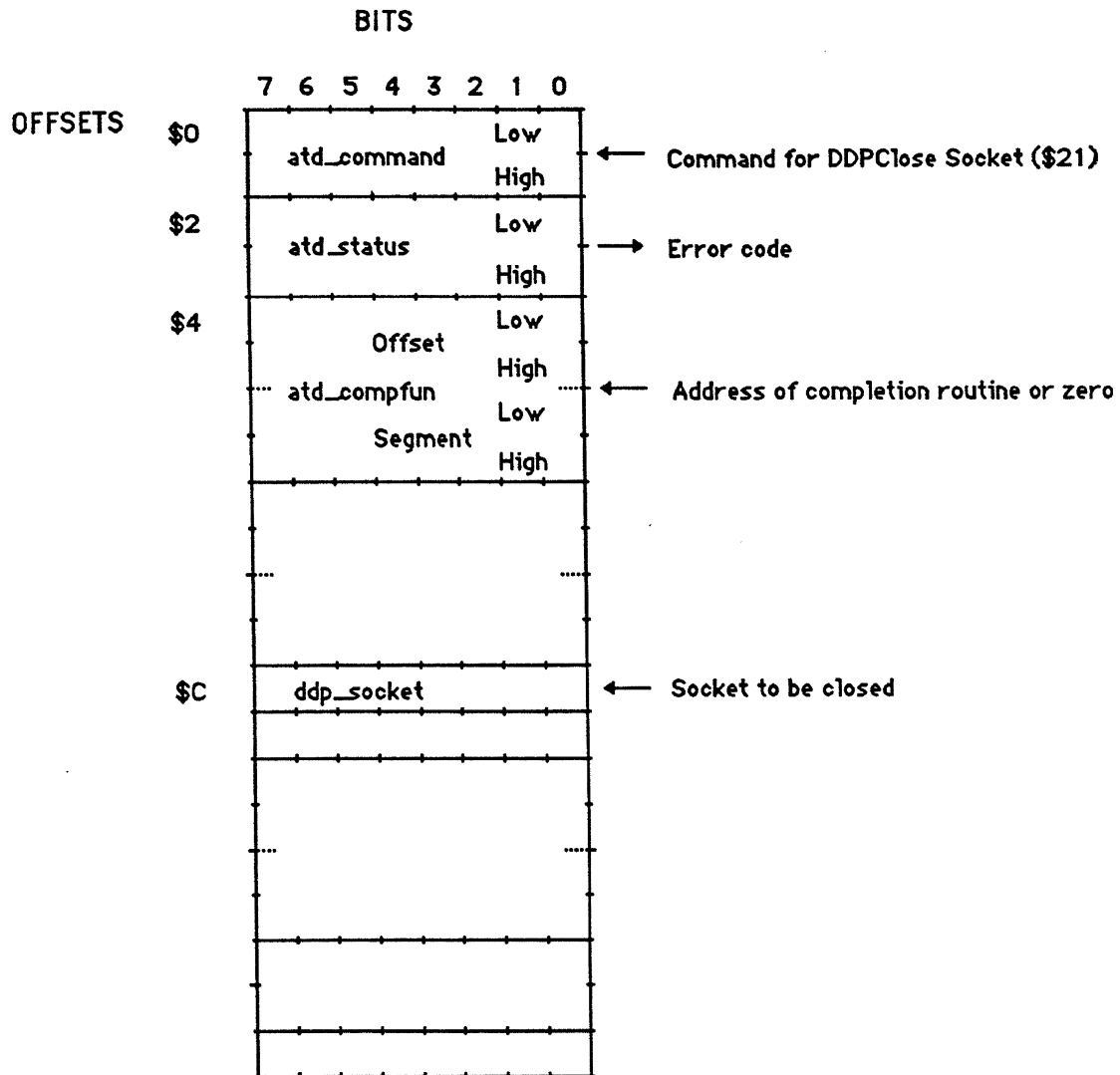


Figure 5-3. DDPCloseSkt Parameter Block

DDPWrite (\$22)

The DDPWrite call sends a Datagram through the passed socket to the socket address in the embedded AddrBlk.

For DDP packets sent to another network, the caller may specify whether a checksum should be computed for the packet. If the caller desires a checksum, the application should set the ddp_checksum field to a non-zero value when the DDPWrite call is made. If the application is not set in this way, no checksum will be computed.

The following structure is used in the interface to DDPWrite:

DDPParams		
word	atd_command;	command code
word	atd_status;	status word/error return
dword	atd_compfun;	completion routine or zero
AddrBlk	ddp_addr;	destination address of packet
byte	ddp_socket;	socket for command
byte	ddp_type;	DDP type for header
dword	ddp_buffptr;	segment:offset of buffer area or socket listener (DDPOpenSkt)
word	ddp_buffsize;	size of buffer area
byte	ddp_chksum;	checksum flag

Parameter usage and error returns for the DDPWrite command are as follows:

Parameter Usage

Input

-->	atd_command	DDPWrite
-->	atd_compfun	address of completion routine or 0
-->	ddp_addr	destination address
-->	ddp_socket	socket to be sent through
-->	ddp_type	DDP type for header
-->	ddp_buffptr	segment:offset of packet data
-->	ddp_buffsize	size of packet data
-->	ddp_chksum	checksum flag

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

```

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
DDP_SKTERR
DDP_LENERR
STACKERROR
NOBRDGERR
MAXCOLISERR
MAXDEFERERR

```

Figure 5-4 illustrates the parameter block for the DDPWrite command.

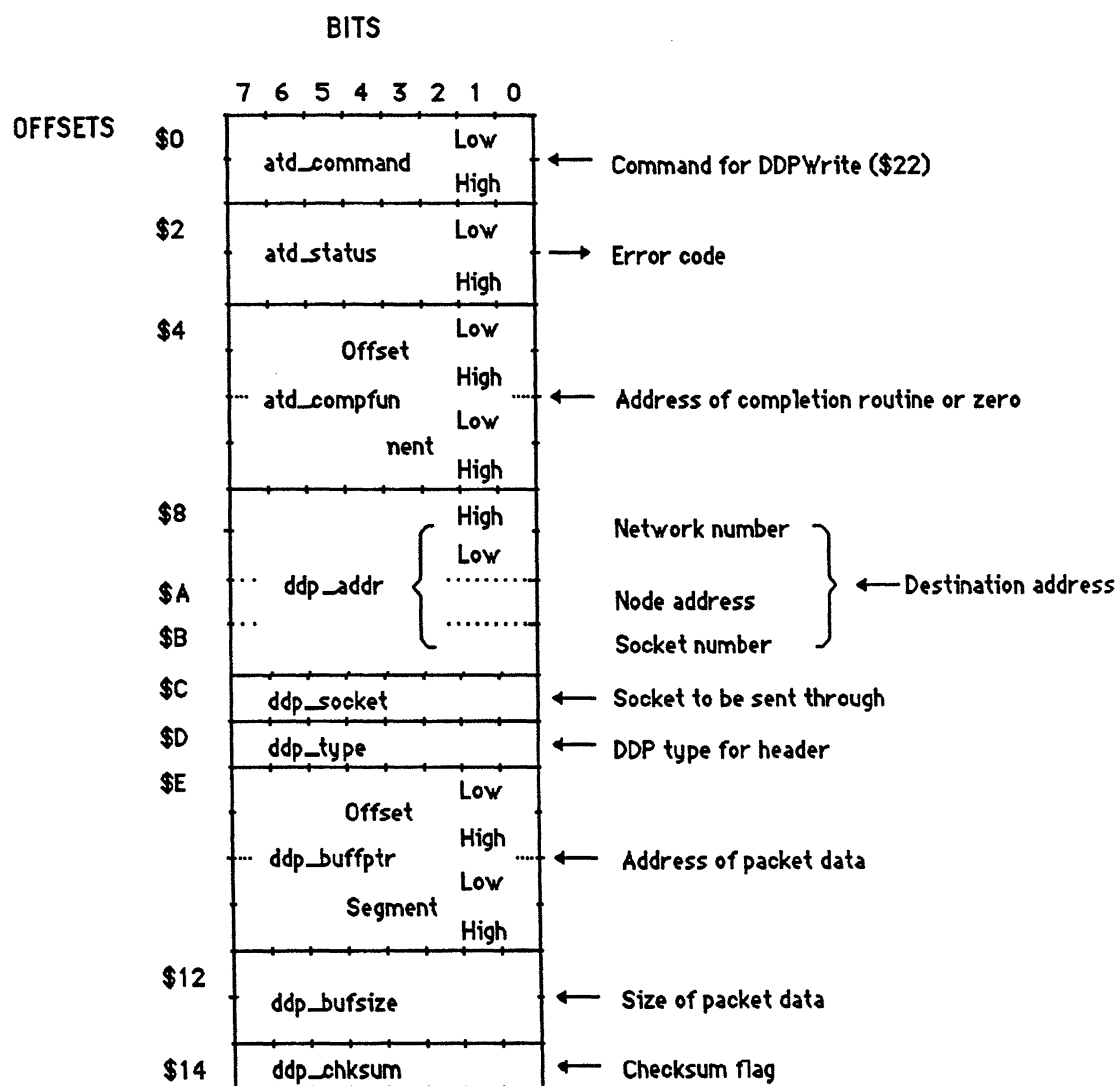


Figure 5-4. DDPWrite Parameter Block

DDPRead (\$23)

The DDPRead command receives a Datagram through the passed socket.

This command uses the default socket listener to read the Datagram, and can be used only if no user socket listener was installed. If the `ddp_buffptr` field is set by the caller, the entire contents of the packet (including the headers) will be placed at the passed buffer. The `ddp_buffsize` field will be updated to reflect the number of bytes placed in the buffer.

There is one special case of this command. If this call is made with the `ddp_buffptr` field set to zero, the driver will read the incoming packet into an internal buffer. The driver will then call the completion routine, which *must* be specified, with the `ddp_buffptr` field set to the *segment:offset* of this internal buffer. The completion routine will then be able to read the packet from this buffer and save the contents locally if the routine chooses to do so.

Note: The case just described is an exception to the calling conventions for completion routines because the routine is called with interrupts disabled, and because the completion routine should *not* enable interrupts until it has safely copied the packet contents to a local buffer.

Parameter usage and error returns for the DDPRead command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	DDPRead
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>ddp_socket</code>	socket to listen on
-->	<code>ddp_buffptr</code>	segment:offset of buffer or zero
-->	<code>ddp_buffsize</code>	size of packet buffer

Output

<--	<code>atd_status</code>	error code
<--	<code>ddp_buffptr</code>	packet buffer (if 0 on input)
<--	<code>ddp_buffsize</code>	size of Datagram at buffer

Error Returns

```

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
DDP_SKTERR
DDP_LENERR
DDP_CANCELLED

```

Figure 5-5 illustrates the parameter block for the DDPRead command.

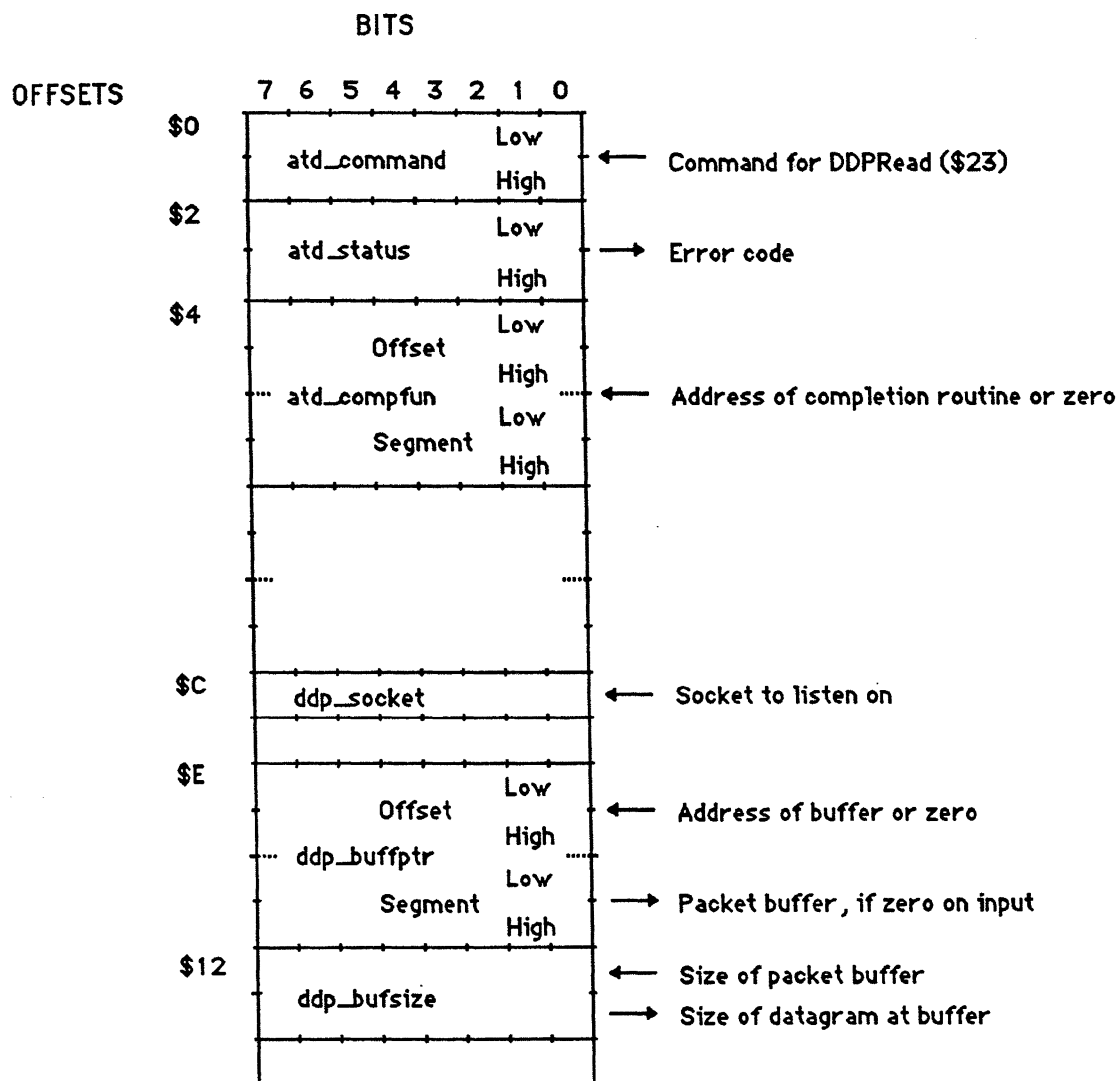


Figure 5-5. DDPRead Parameter Block

DDPCancel (\$24)

The DDPCancel command cancels any outstanding DDP command. The parameter structure pointed to by the ddp_buffptr field must be the same one as that passed in with the command being cancelled.

Parameter usage and error returns for the DDPCancel command are as follows:

Parameter Usage

Input

-->	atd_command	DDPCancel
-->	atd_compfun	address of completion routine or 0
-->	ddp_buffptr	params for command being cancelled

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER

Figure 5-6 illustrates the parameter block for the DDPCancel command.

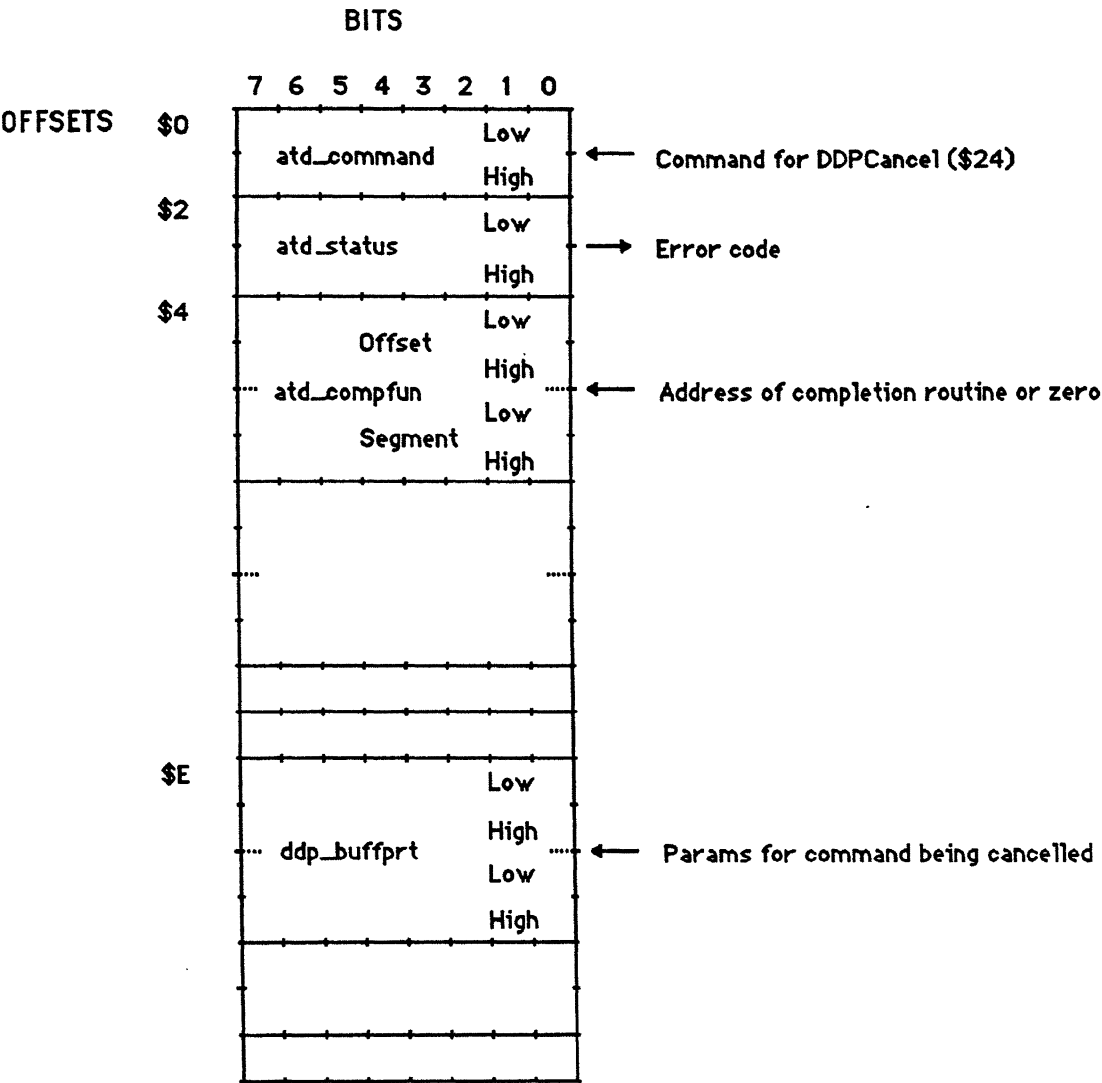


Figure 5-6. DDPCancel Parameter Block

Chapter 6

Name Binding Protocol (NBP) Commands

Introduction

The Name Binding Protocol (NBP) commands allow an application to look up and register names on the network. NBP commands include

- NBPSRegister
- NBPRemove
- NBPLookup
- NBPSConfirm
- NBPSCancel

In all cases, the NBP calls will return zero (NOERR) if the call is completed successfully, or a negative number if there is an error.

For calls made asynchronously (called with `async` equal to one), it is important that the caller not modify the parameter blocks that it passes to NBP until the command has been completed. The applications may monitor the status word of the passed parameter structure for completion. Until the call is complete, this status word will have a positive value. Upon completion, the status word will be zero if there were no errors, or a negative number if there was an error.

Note: Some of the NBP commands take retry interval units as input. The length of these units of time is one second.

Protocol Parameter Structures

The NBP uses structures called “tuples.” A network entity is fully specified by a tuple. Tuples consist of

- a full network address (`AddrBlk`)
- an enumerator
- an entity name

An entity name is a set of three packed Pascal-style character strings each containing up to 32 characters that identify the object, the type of the object (such as the LaserWriter), and the zone (a collection of networks) of the object. An example of an entity name is shown as follows:

```
db 5, 'Dante', 11, 'LaserWriter', 1, '*'
```


Pascal-style character strings consist of one byte indicating the length of the string (exclusive of the length byte), followed by the bytes that make up the character string; thus an entity name's length may be up to 99 bytes long.

```

NBPTuple
    AddrBlk      tup_address;      network address of entity
    byte         tup_enum;         enumerator field
    char[99]     tup_entname;      space for EntityName
    
```

Figure 6-1 illustrates the NBPTuple parameter block.

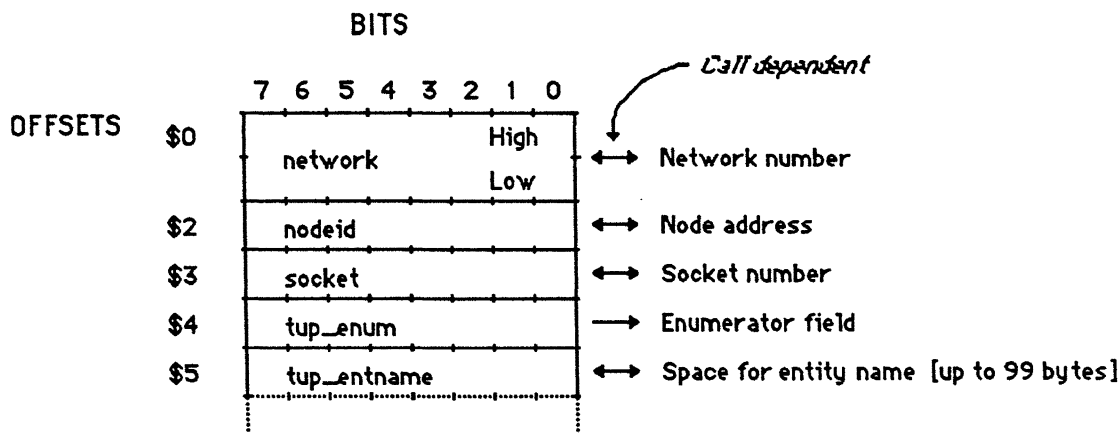


Figure 6-1. NBPTuple Parameter Block

The NBP manages a table of entity names associated with a node. Provided to NBP by the caller upon registering a name, the structures are just a linked list of tuples as follows:

```

NBPTabEntry
    dword      tab_nxtentry;      segment:offset of next entry
    NBPTuple   tab_tuple;        actual tuple for this entry
    
```

Figure 6-2 illustrates the NBPTabEntry parameter block.

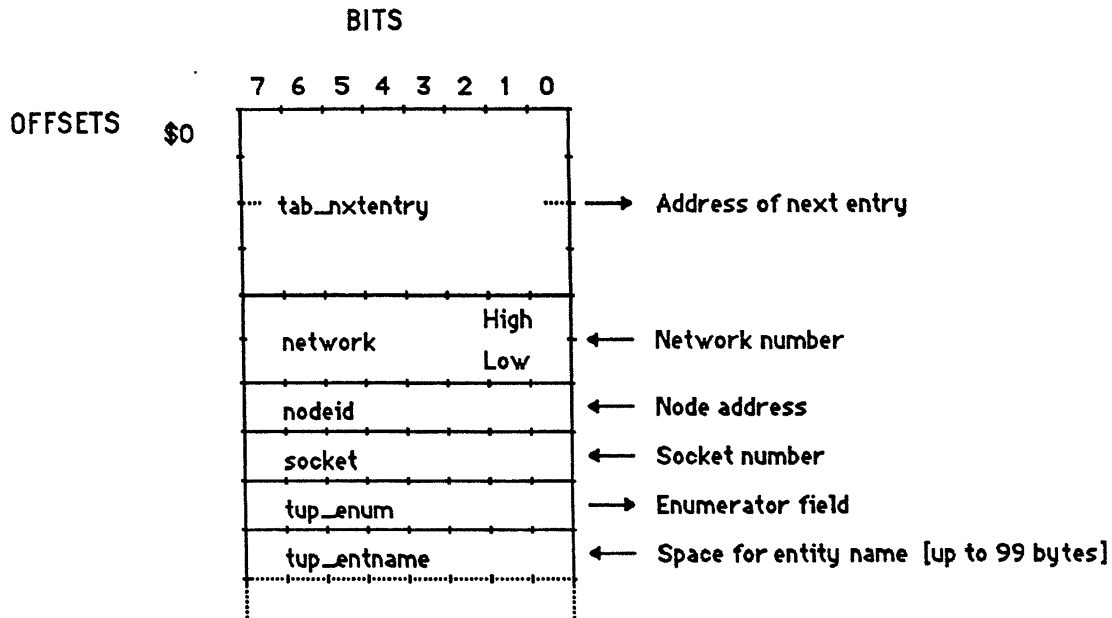


Figure 6-2. NBPTabEntry Parameter Block

The following parameter structure is used in the interface to the Name Binding Protocol routines:

NBPParams		
word	atd_command;	command code
word	atd_status;	status word/error return
dword	atd_compfun;	completion routine or zero
AddrBlk	nbp_addr;	for confirm, original address
word	nbp_toget;	number of matches to get were found
dword	nbp_buffptr;	segment:offset of buffer area
word	nbp_buffsize;	size of buffer area
byte	nbp_interval;	interval (seconds) between tries
byte	nbp_retry;	number of tries
dword	nbp_entname;	segment:offset of EntityName

Figure 6-3 shows the parameter structure used in the interface to the NBP routines.

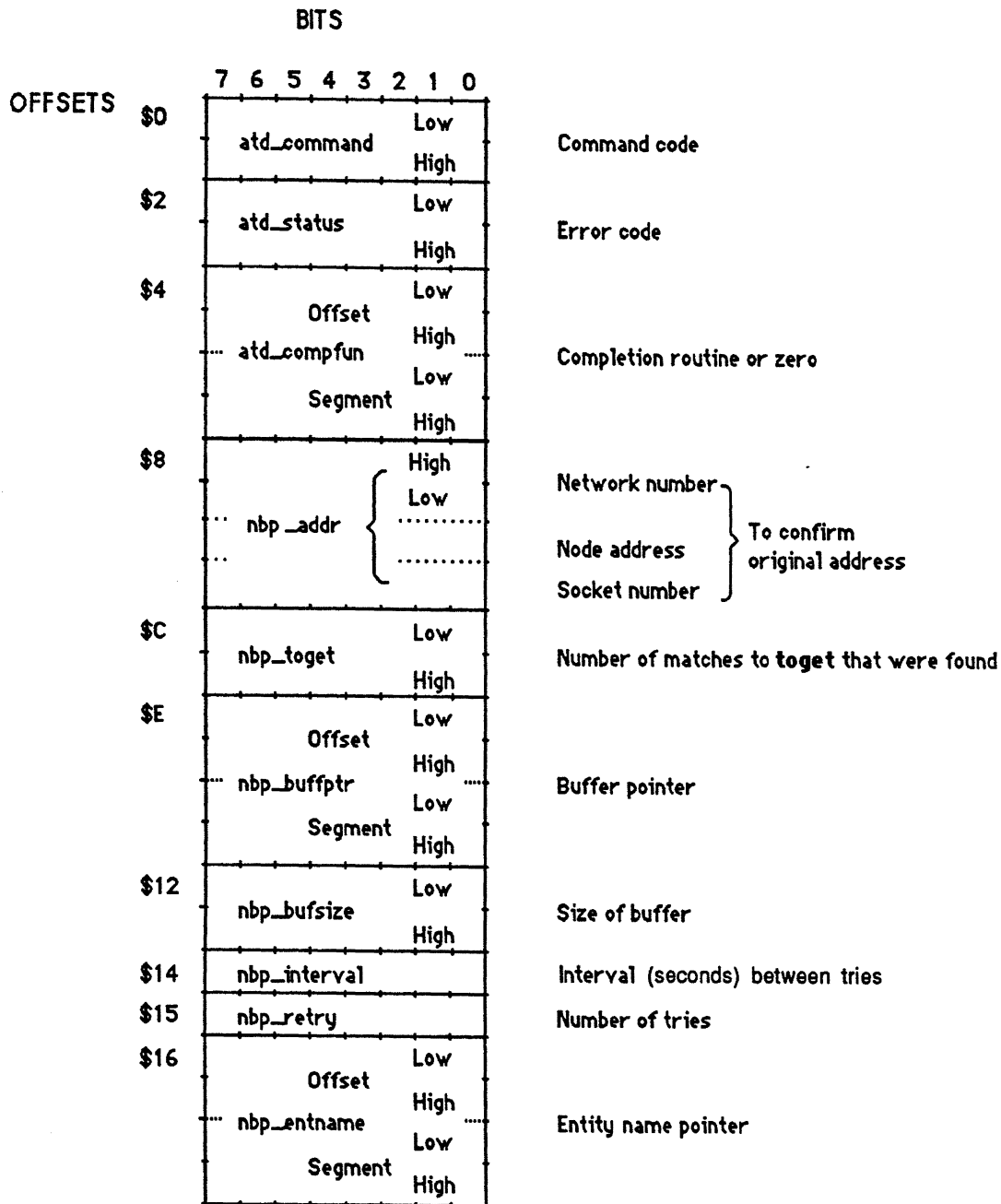


Figure 6-3. General Parameter Block for Name Binding Protocol (NBP) Commands

NBPRegister (\$30)

The NBPRegister command adds an entity to the local node "names table."

For this call, the `nbp_buffptr` field of the `NBPParams` structure is actually a pointer to a `NBPTabEntry` filled in by the caller. This field is the actual memory location that is used by the other NBP routines, so the caller must not modify it until it has been unregistered (by using a `NBPRemove` call). The `nbp_retry` and `nbp_interval` fields should contain information specifying the number of retries and at what time intervals retries should be sent. The zone portion of the registered name should be the single character "*" (preceded by a 1 for length of course).

Parameter usage and error returns for the NBPRegister command are as follows:

Parameter Usage

Input

--> <code>atd_command</code>	<code>NbpRegister</code>
--> <code>atd_compfun</code>	address of completion routine or 0
--> <code>nbp_buffptr</code>	segment:offset of <code>NBPTabEntry</code>
--> <code>nbp_interval</code>	time to wait between lookups
--> <code>nbp_retry</code>	number of times to send lookups

Output

<-- <code>atd_status</code>	error code
-----------------------------	------------

Error Returns

`NOERR`
`ATNOTINITIALIZED`
`BAD_PARAMETER`
`NO_MEM_ERROR`
`NAME_IN_USE`
`NBP_CANCELLED`

Figure 6-4 illustrates the parameter block for the NBPRegister command.

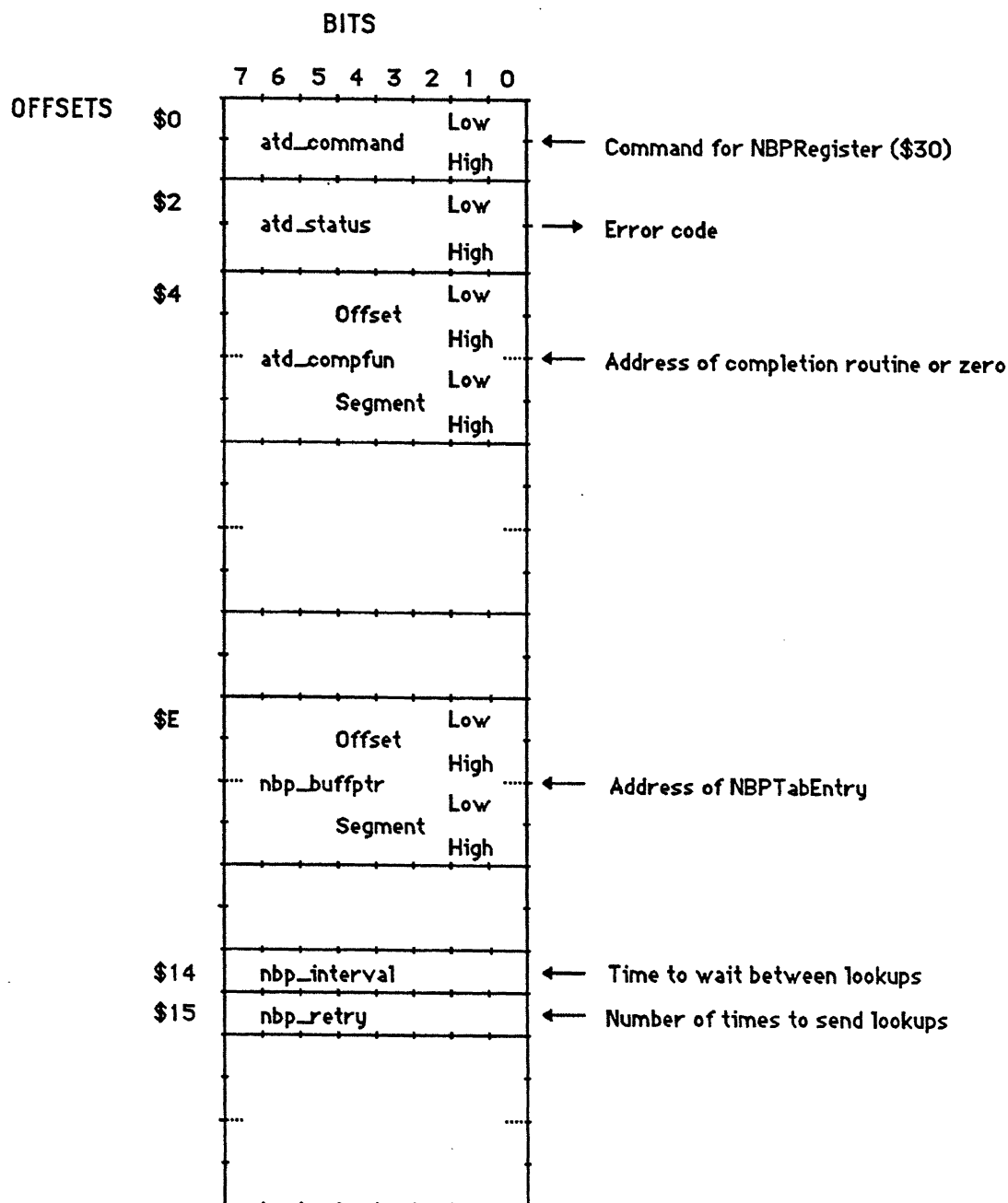


Figure 6-4. NBPRegister Parameter Block

NBPRemove (\$31)

The NBPRemove call removes the local names table entry associated with the passed entity name. The name should be in the form of an EntityName. If there is an error, a negative value is returned. If there is no error, a pointer to the NBPTabEntry structure that was used to register the passed name is returned.

Parameter usage and error returns for the NBPRemove command are as follows:

Parameter Usage

Input

-->	atd_command	NBPRemove
-->	atd_compfun	address of completion routine or 0
-->	nbp_entname	EntityName to remove

Output

<--	atd_status	error code
<--	nbp_buffptr	segment:offset of NBPTabEntry

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NBP_NOTFOUND
NBP_CANCELLED

Figure 6-5 illustrates the parameter block for the NBPRemove command.

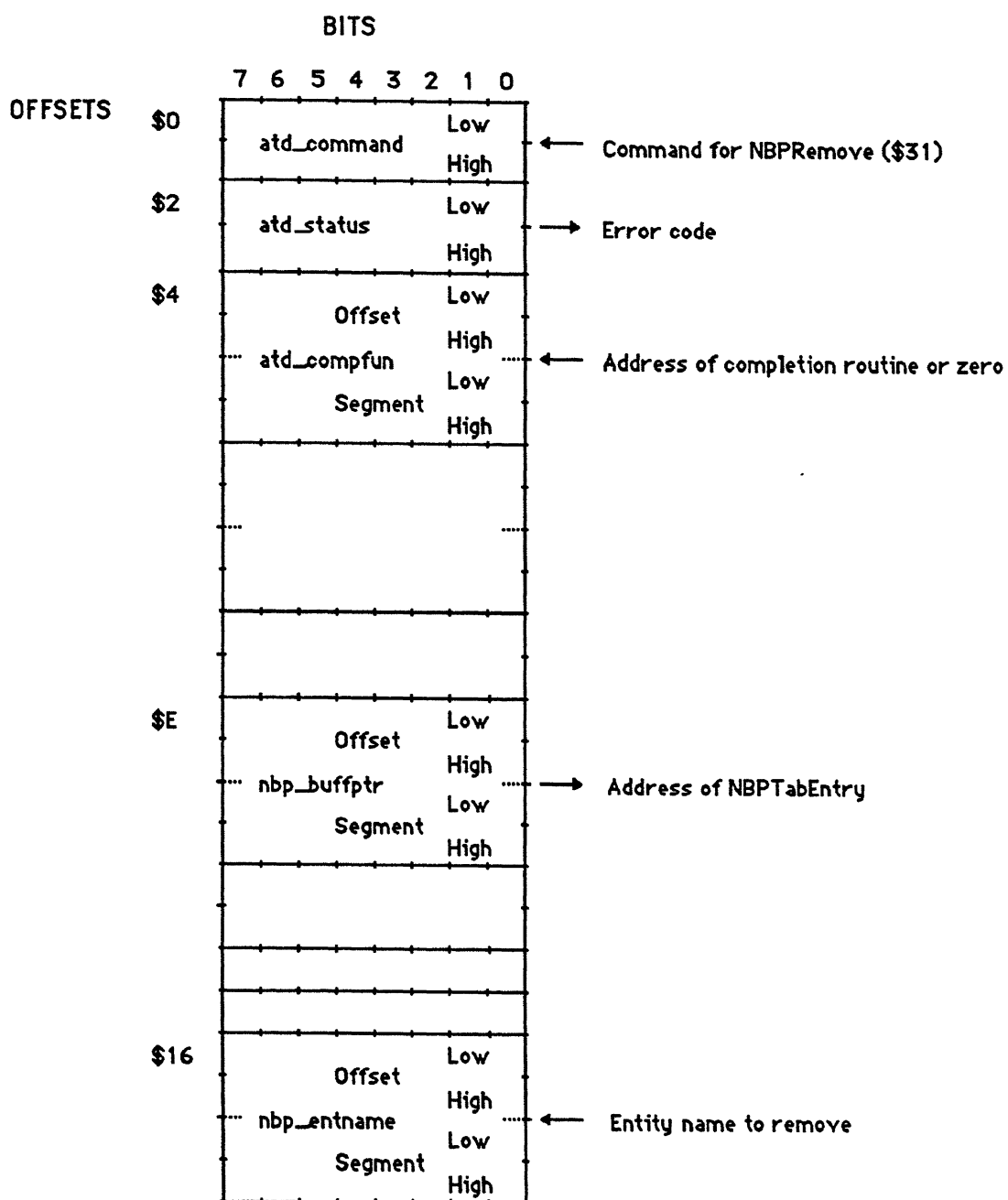


Figure 6-5. NBPRemove Parameter Block

NBPLookup (\$32)

The NBPLookup call searches the network or networks for entities that match (allowing wildcards) the entity name contained in the `nbp_entname` field in the parameter structure.

The name in the `nbp_entname` field should be in the form of an `EntityName`. NBP will place up to `nbp_toget` tuples at the address passed by the caller in the `nbp_buffptr` field.

After the call is completed, the `nbp_toget` field will provide the number of matching entities found, which is the number of valid tuples at the `nbp_buffptr` field. If more matches were found than could be fit into the buffer, the `nbp_status` field will have the value `NBP_NOROOM` and the `nbp_toget` field indicate the number that fit into the buffer.

If the buffer size is four or greater, but not enough to hold the entity name, the address of the entity that is being looked up will be stored in the buffer and a `NBP_NO_ROOM` error returned. This allows a lookup for a nonwildcard name to return only the address and therefore use a very small buffer.

Parameter usage and error returns for the NBPLookup command are as follows:

Parameter Usage

Input

--> <code>atd_command</code>	NBPLookup
--> <code>atd_compfun</code>	address of completion routine or 0
--> <code>nbp_toget</code>	number of matches to get
--> <code>nbp_buffptr</code>	segment:offset of buffer for responses
--> <code>nbp_buffsize</code>	size of buffer at <code>nbp_buffptr</code>
--> <code>nbp_interval</code>	time to wait between lookups
--> <code>nbp_retry</code>	number of times to send lookups
--> <code>nbp_entname</code>	<code>EntityName</code> to look up

Output

<-- <code>atd_status</code>	error code
<-- <code>nbp_toget</code>	number of matches in the buffer

Error Returns

`NOERR`
`ATNOTINITIALIZED`
`BAD_PARAMETER`
`NBP_NOTFOUND`
`NBP_NO_ROOM`
`NO_MEM_ERROR`
`NBP_CANCELLED`

Figure 6-6 illustrates the parameter block for the NBPLookup command.

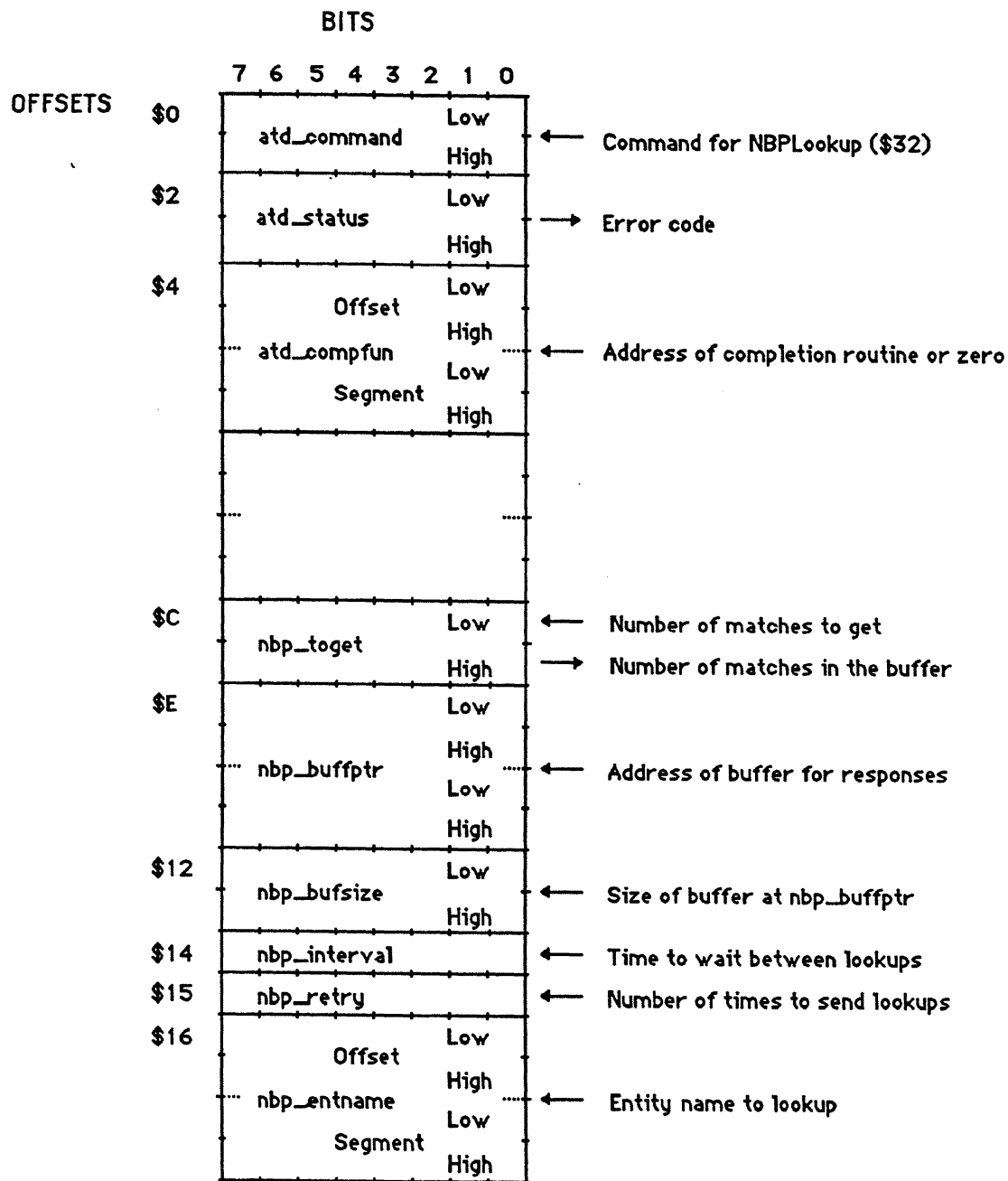


Figure 6-6. NBPLookup Parameter Block

NBPCConfirm (\$33)

The NBPCConfirm call makes sure that a previously known entity is still active on the network.

For this call, the `nbp_addr` field in the parameter structure holds the previous network address of the entity in question. In addition, the `nbp_entname` field contains an entity name of the form in NBPLookup.

If the entity is no longer on the network, the `atd_status` field will be `NBP_NOCONFIRM` upon completion. If the entity exists but has a different socket, the status field will be `NBP_NEWSOCKET`, and the `nbp_addr` field will have the updated network address. However, if the entity is still active at the old network address, the `atd_status` field will be zero. If the address and NODEID is zero or the programmer's own address, the confirm will be done in the internal names registered by the driver.

Parameter usage and error returns for the NBPCConfirm command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	NBPCconfirm
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>nbp_addr</code>	old address (being confirmed)
-->	<code>nbp_interval</code>	time to wait between lookups
-->	<code>nbp_retry</code>	number of times to send
-->	<code>nbp_entname</code>	EntityName to confirm

Output

<--	<code>atd_status</code>	error code
<--	<code>nbp_addr</code>	new address of entity

Error Returns

```

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NBP_NEWSOCKET
NBP_NOCONFIRM
NBP_CANCELLED
    
```

Figure 6-7 illustrates the parameter block for the NBPCConfirm command.

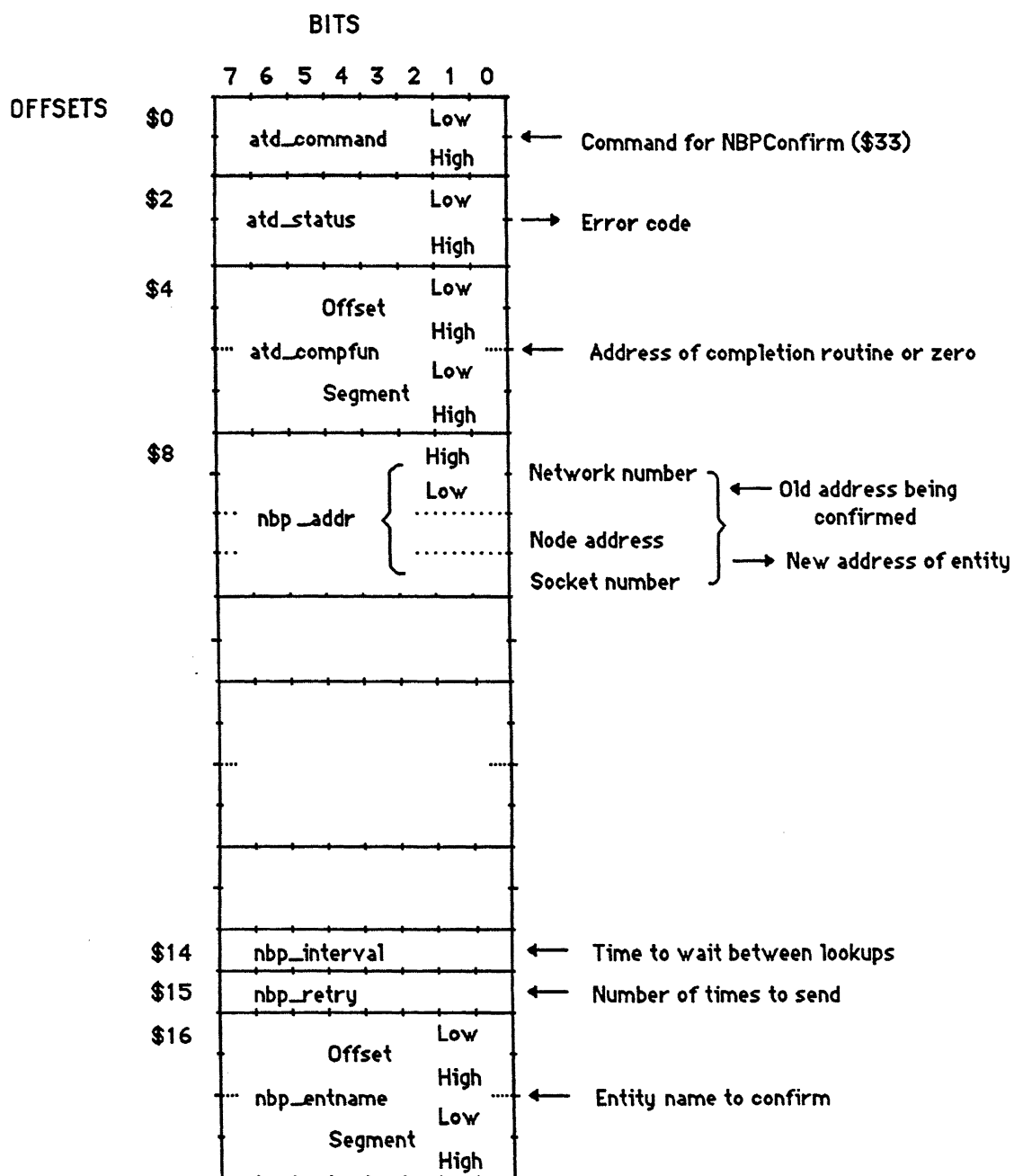


Figure 6-7. NBPCConfirm Parameter Block

NBPCancel (\$34)

The NBPCancel call cancels any outstanding NBP command.

This call takes as input the address of the parameter block associated with the command. If the command being cancelled had a completion routine, the application calls the routine with the `atd_status` field set to `NBP_CANCELLED`.

Parameter usage and error returns for the NBPCancel command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	NBPCancel
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>nbp_buffptr</code>	params of command being cancelled

Output

<--	<code>nbp_status</code>	error code
-----	-------------------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER

Figure 6-8 illustrates the parameter block for the NBPCancel command.

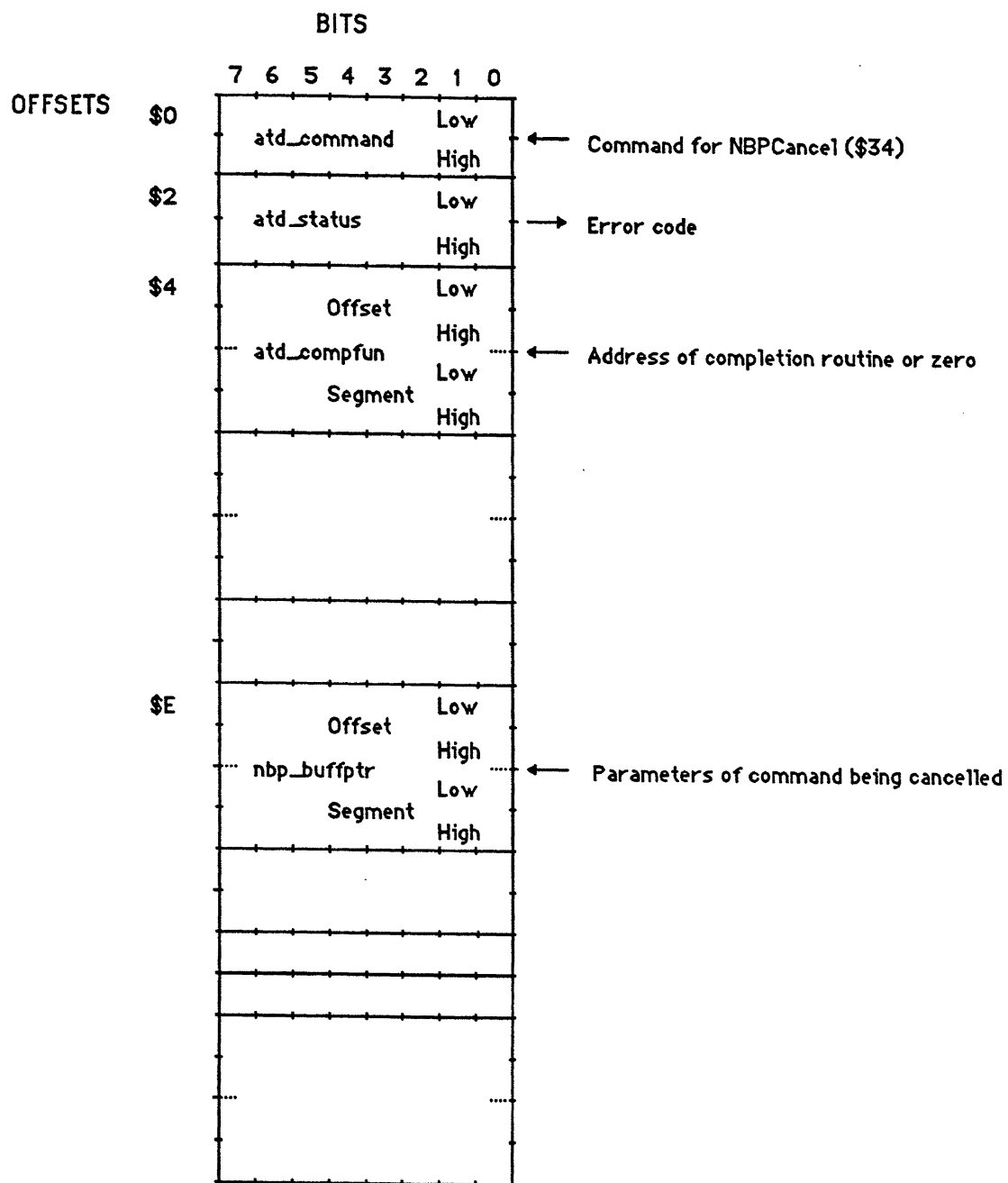


Figure 6-8. NBPCancel Parameter Block

Chapter 7

Zone Information Protocol (ZIP) Commands

Introduction

The Zone Information Protocol (ZIP) commands provide the names of all zones on the network to an application. ZIP commands include

- ZIPGetZoneList
- ZIPGetMyZone

In all cases, the ZIP calls will return zero (NOERR) if the call is completed successfully, or a negative number if there is an error.

For calls made asynchronously, it is important that the caller not modify the parameter blocks passed to ZIP until the command has been completed. The status word of the passed parameter structure may be monitored for completion. Until the call is complete, this status word will have a positive value. Upon completion, that word will be zero if there were no errors, or a negative number if there was an error.

Protocol Parameter Structures

The following parameter structure is used in the interface to the ZIP commands.

ZIPParams	
word atd_command;	command code
word atd_status;	status word/error return
dword atd_compfun;	completion routine or zero
dword zip_fill;	filler dword
word zip_zones;	number of zone names in returned list
dword zip_buffptr;	segment:offset of buffer area
word zip_buffsize;	size of buffer area

Figure 7-1 illustrates the parameter structure used in the interface to the ZIP commands.

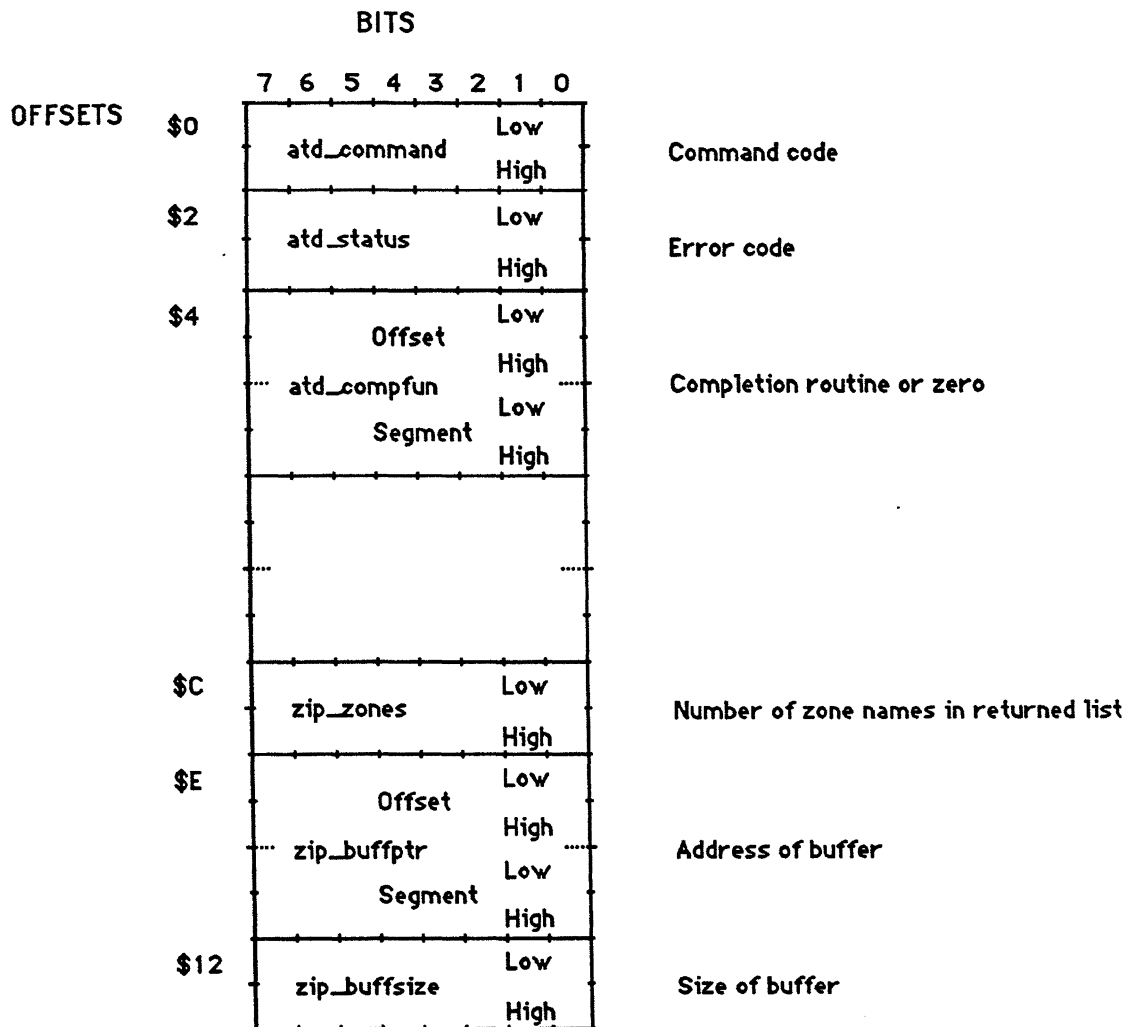


Figure 7-1. General Parameter Block for Zone Information Protocol (ZIP) Commands

ZIPGetZoneList (\$35)

The ZIPGetZoneList call returns a list of all zones on an Internet. The list is returned in the buffer passed by the zip_buffptr field, and the number of names in the list is returned in the zip_zones field when the call completes. The list is made up of packed Pascal-style character strings (length byte followed by the characters of the name). If there were more names than could fit in zip_buffsize bytes, an error is returned; however, as many names as can fit are placed in the buffer.

Parameter usage and error returns for the ZIPGetZoneList are as follows:

Parameter Usage

Input

-->	atd_command	ZIPGetZoneList
-->	atd_compfun	address of completion routine or 0
-->	zip_buffptr	segment:offset of names buffer
-->	zip_buffsize	size of buffer at zip_buffptr

Output

<--	atd_status	error code
<--	zip_zones	number of zone names in list

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
ATP_OVERFLOW

Figure 7-2 illustrates the parameter block for the ZIPGetZoneList command.

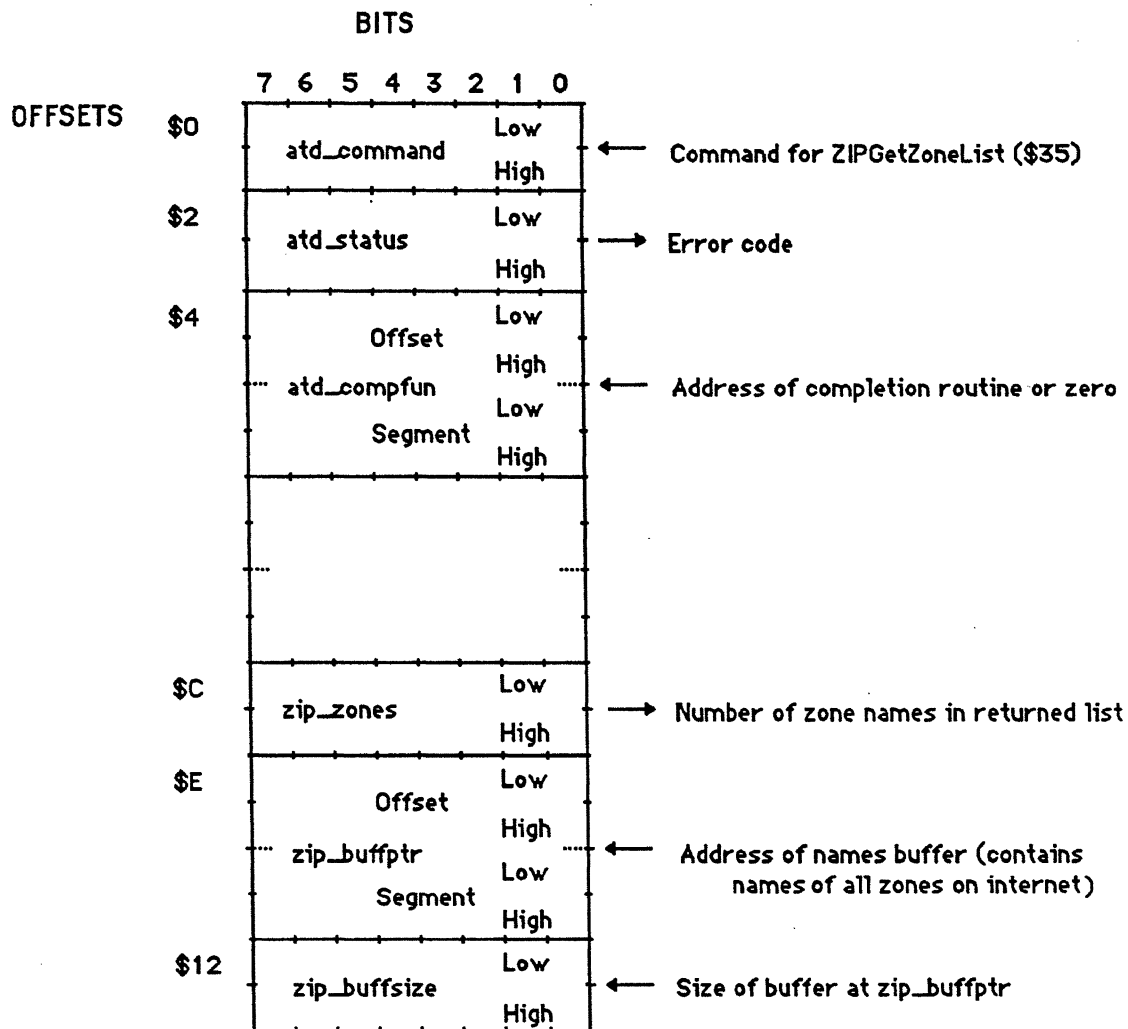


Figure 7-2. ZIPGetZoneList Parameter Block

ZIPGetMyZone (\$36)

The ZIPGetMyZone call returns the name of the caller's zone. The name of the local zone is returned in the buffer passed by the zip_buffptr field. The name is in the form of a Pascal-style character string.

Parameter usage and error returns for the ZIPGetZoneList are as follows:

Parameter Usage

Input

-->	atd_command	ZIPGetMyZone
-->	atd_compfun	address of completion routine or 0
-->	zip_buffptr	segment:offset of names buffer
-->	zip_buffsize	size of buffer at zip_buffptr

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
ATP_OVERFLOW
BAD_PARAMETER

Figure 7-3 illustrates the parameter block for the ZIPGetMyZone command.

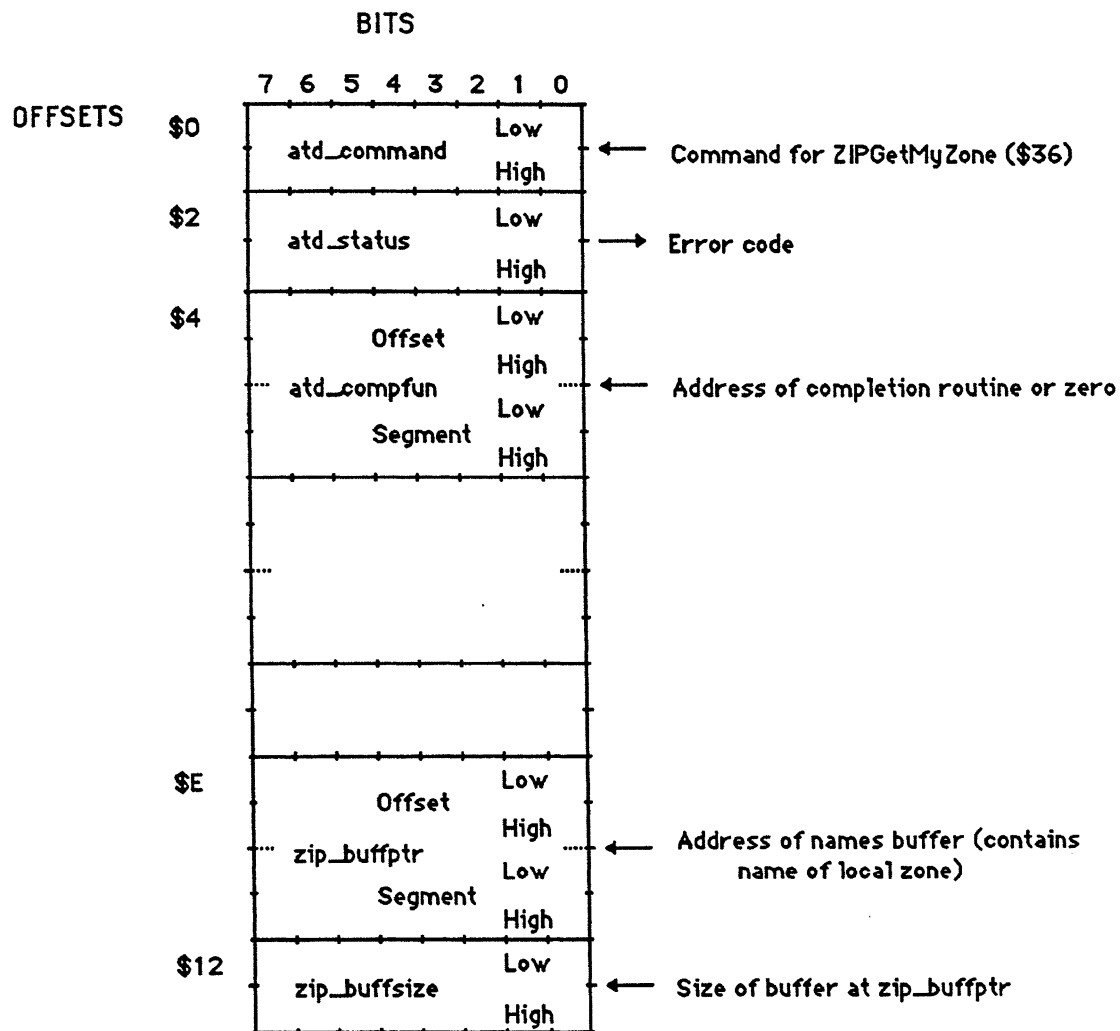


Figure 7-3. ZIPGetMyZone Parameter Block

Chapter 8

AppleTalk Transaction Protocol (ATP) Commands

Introduction

The AppleTalk Transaction Protocol (ATP) commands manage request and response transactions between two network entities used by higher-level protocols. The ATP commands include

- ATPOpenSocket
- ATPCloseSocket
- ATPSendRequest
- ATPGetRequest
- ATPSendResponse
- ATPAddResponse
- ATPCancelTrans
- ATPCancelResponse
- ATPCancelRequest

In all cases, the ATP calls will return zero (NOERR) if the call is completed successfully, or a negative number if there is an error.

For calls made asynchronously, it is important that the caller not modify the parameter blocks passed to ATP until the command has been completed. The status word of the passed parameter structure may be monitored for completion. Until the call is complete, this status word will have a positive value. Upon completion, that word will be zero if there were no errors, or a negative number if there was an error.

Protocol Parameter Structures

The ATP implementation uses Buffer Data Structure (BDS) for storing the incoming response packets/data associated with an outgoing request. A BDS is actually an array (from one to eight elements) of structures, described as follows:

BDSElement		
dword	bds_buffptr;	segment:offset of data buffer
word	bds_buffsize;	size of data buffer in bytes
word	bds_datasize;	amount of data put in buffer
char (4)	bds_usrbytes;	four bytes for ATP header

Figure 8-1 illustrates the BDS entry format.

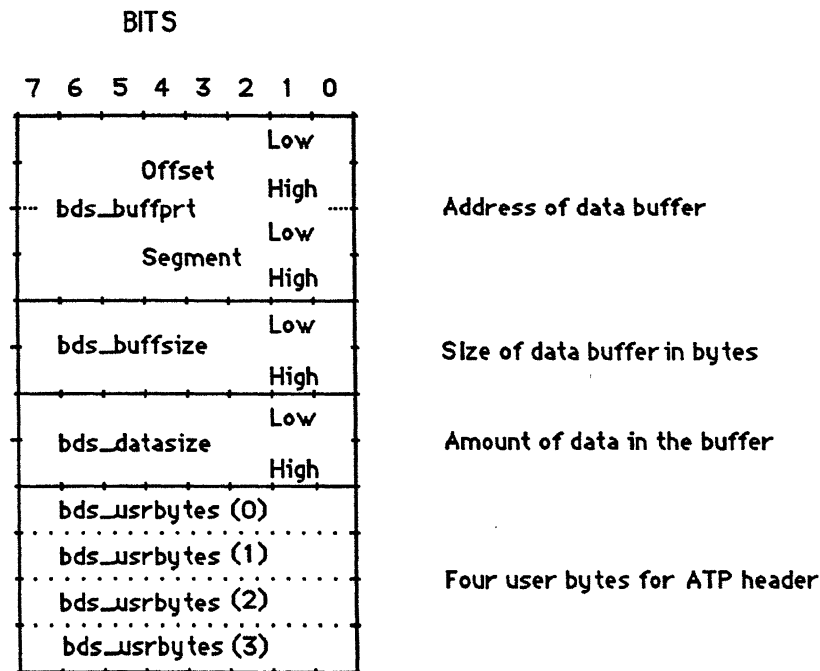


Figure 8-1. BDS Entry Format

The following parameter structure is used by the ATP protocol interface.

ATPPParams		
word	atd_command;	command code
word	atd_status;	status word/error return
dword	atd_compfun;	completion routine or zero
AddrBlk	atp_addr;	embedded address block
byte	atp_socket;	socket associated with this transaction
byte	atp_fill;	filler byte
dword	atp_buffptr;	segment:offset of request data buffer
word	atp_buffsize;	amount of data at above address
byte	atp_interval;	interval (seconds) between tries
byte	atp_retry;	number of times to try request
byte	atp_flags;	control info (XO, etc.) (NoRelease = 02H)
byte	atp_seqbit;	bitmap, sequence info
word	atp_tranid;	transaction ID
char[4]	atp_userbytes;	four user bytes for ATP header
byte	atp_bdsbuffs;	number of BDSElements at atp_bdsptr
byte	atp_bdsresps;	number of responses at atp_bdsptr
dword	atp_bdsbptr;	segment:offset of BDS for responses

Figure 8-2 illustrates the general parameter structure used in the interface to the ATP commands.

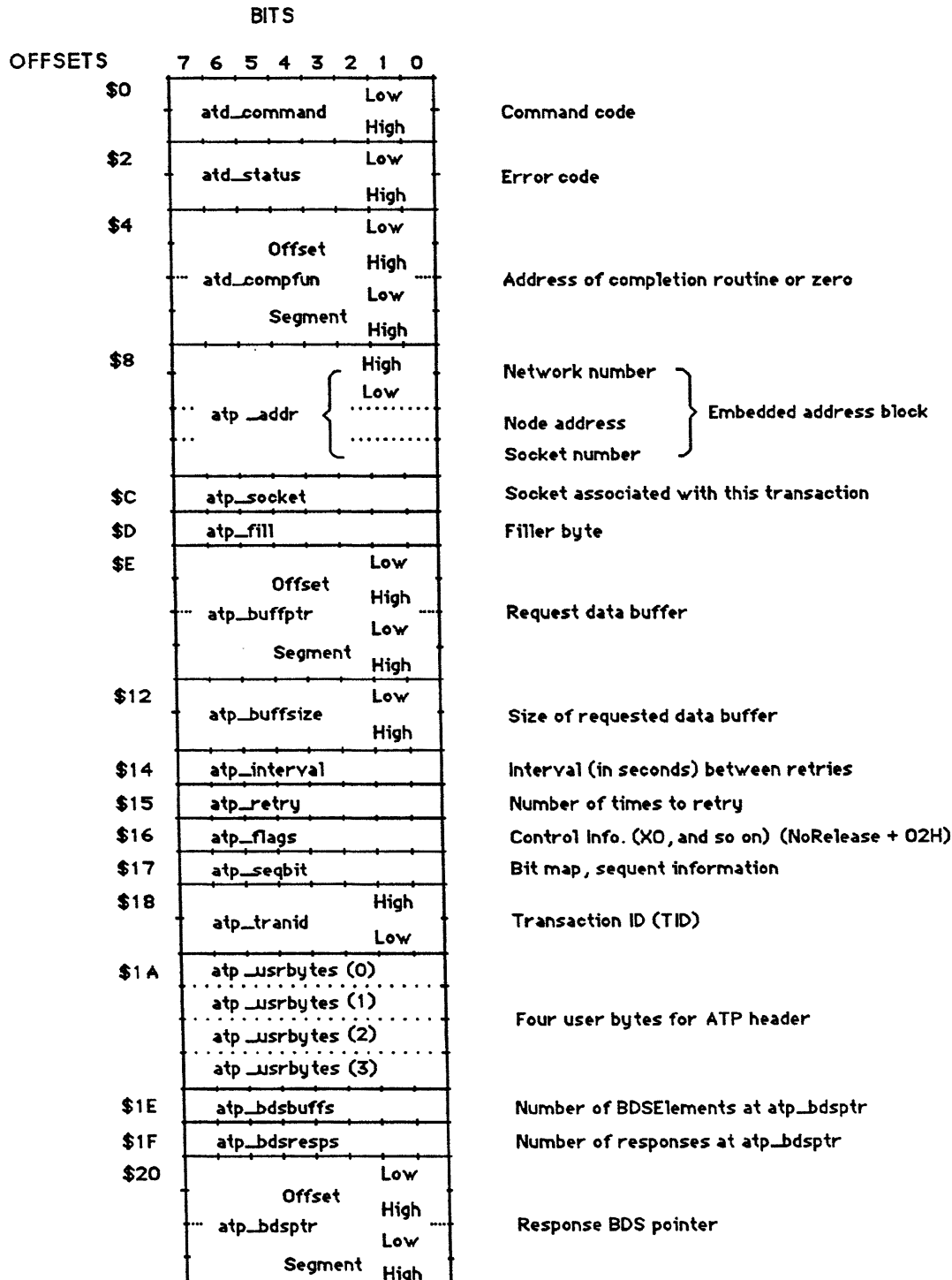


Figure 8-2. General Parameter Block for AppleTalk Transaction Protocol (ATP) Commands

ATPOpenSocket (\$40)

The ATPOpenSocket command opens a socket for the purpose of receiving requests.

If not zero, the number specified by the atp_socket field is used as the socket number. If that socket number is already being used or is between 128 and 254, an error is returned. If the atp_socket field equals zero and a socket number is available, then the driver allocates a socket number dynamically and returns the value.

The AddrBlk pointed to by the atp_addr field allows a socket to be opened that receives requests only from matching network addresses. If any of the fields of that AddrBlk are zero, no filtering will be done on that part of the address (network, node, or socket).

Parameter usage and error returns for the ATPOpenSocket command are as follows:

Parameter Usage

Input

-->	atd_command	ATPOpenSocket
-->	atd_compfun	address of completion routine or 0
-->	atp_addr	address for filtering requests
-->	atp_socket	suggested socket number

Output

<--	atd_status	error code
<--	atp_socket	actual socket number

Error Returns

ATNOTINITIALIZED
BAD_PARAMETER
NO_MEM_ERROR
DDP_SKTERR

Figure 8-3 illustrates the parameter block for the ATPOpenSocket command.

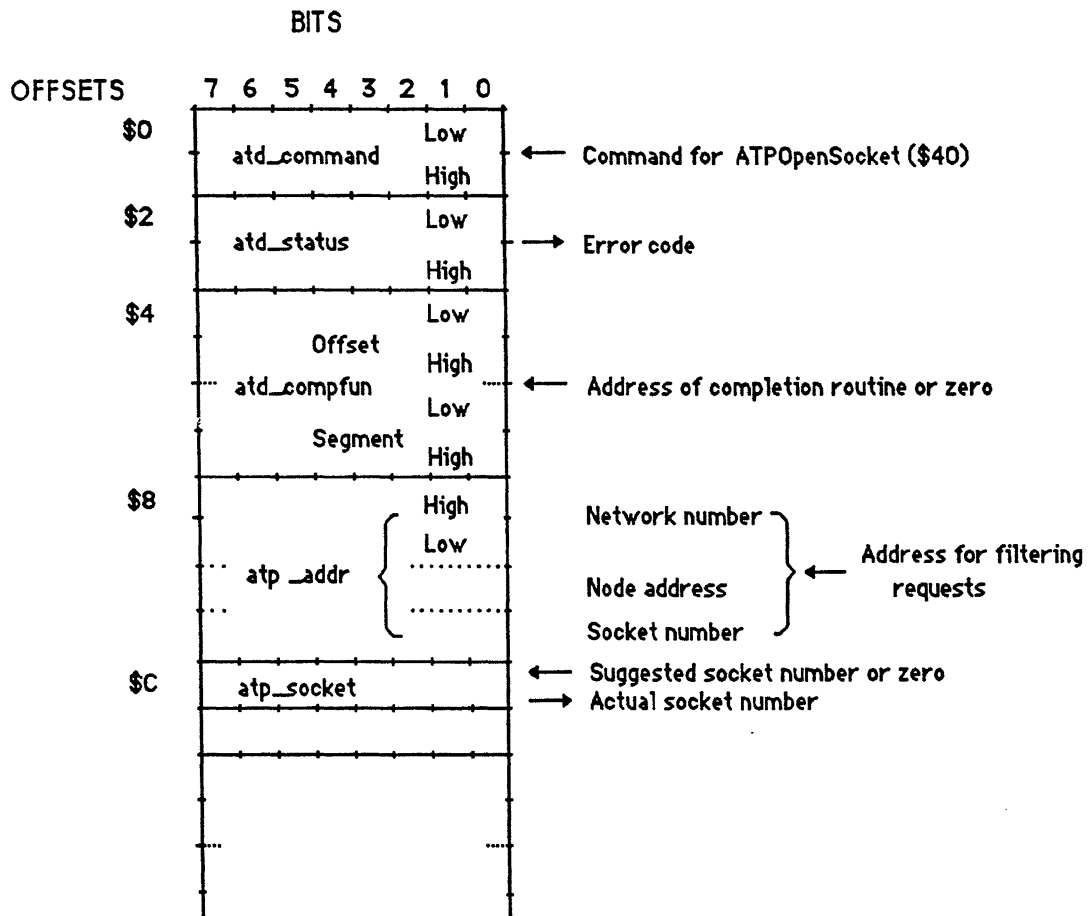


Figure 8-3. ATPOpenSocket Parameter Block

ATPCloseSocket (\$41)

The ATPCloseSocket command closes a socket opened for receiving requests. Any outstanding ATP commands (ATPGetRequest or ATPSendResponse) on that socket will be cancelled. ATPCloseSocket cannot close socket 1 (RTMP's socket) or socket 2 (NBP's socket). Parameter usage and error returns for the ATPCloseSocket command are as follows:

Parameter Usage

Input

```
--> atd_command      ATPCloseSocket
--> atd_compfun      address of completion routine or 0
--> atp_socket        socket to be closed
```

Output

```
<-- atd_status        error code
```

Error Returns

```
NOERR
ATNOTINITIALIZED
BAD_PARAMETER
ATP_BADSOCKET
```

Figure 8-4 illustrates the parameter block for the ATPCloseSocket command.

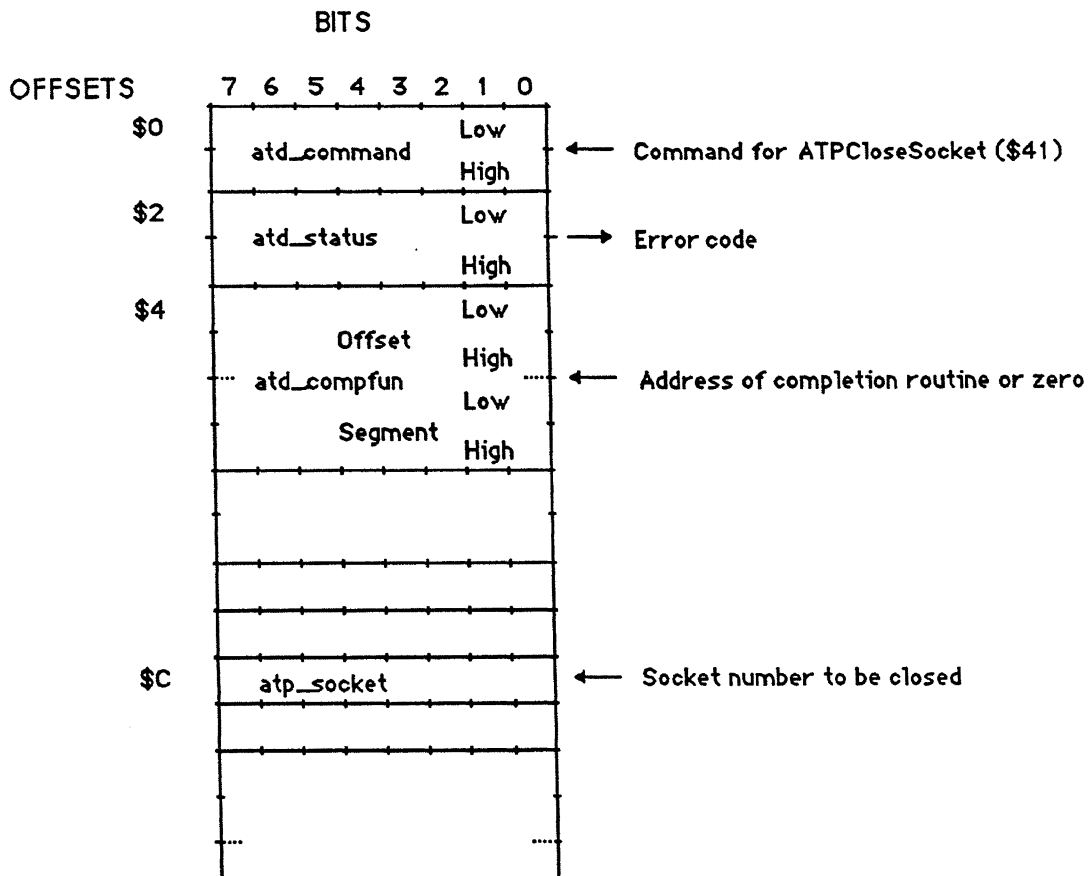


Figure 8-4. ATPCloseSocket Parameter Block

ATPSendRequest (\$42)

The ATPSendRequest call sends a request to the address in the `atp_addr` field. The `atp_buffptr` field contains the address of the data to be sent with the request, and the `atp_buffsize` field gives the number of bytes.

The `atp_bdsptr` field points to a BDS (array of BDSElements) that indicates where the data from incoming response packets should be placed. The `atp_bdsbuffs` field indicates the number of entries in the BDS (and implicitly the number of responses expected), whereas the `atp_bdsresps` field, upon completion, will indicate how many response packets were received (how many of the BDSElements are used).

The `atp_flags` byte should be set by the caller if the transaction should be an “exactly once” transaction (the `XO` bit value is 20H) or if the caller wants a checksum to be calculated for packets on an Internet (the `CHECKSUM` bit value is 1H), or if both conditions exist. The bytes in the `atp_userbytes` field will be placed in the four bytes of the ATP packet’s header. The `atp_compfun` field, if nonzero, should contain the address of code to be executed when the transaction has been completed. The `atp_retry` and `atp_interval` fields indicate how often and at what interval (in seconds) the request will be transmitted. The special case of the `atp_retry` field being set to zero implies an “infinite retry.”

If the `NO_RELEASE` bit and the `XO` bit are set, an ATP release packet will not be sent on the network.

When the ATPSendRequest command returns to the caller, the `atp_tranid` field will have a valid transaction ID for this request.

Parameter usage and error returns for the ATPSendRequest command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	ATPSendRequest
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>atp_addr</code>	network address to send request
-->	<code>atp_buffptr</code>	segment:offset of data for request
-->	<code>atp_buffsize</code>	number of bytes at <code>atp_buffptr</code>
-->	<code>atp_interval</code>	time to wait between sends
-->	<code>atp_retry</code>	number of times to send request
-->	<code>atp_flags</code>	<code>XO</code> bit set for exactly once, CHECKSUM bit may be set
-->	<code>atp_userbytes</code>	four user bytes for ATP header
-->	<code>atp_bdsbuffs</code>	number response packets expected
-->	<code>atp_bdsptr</code>	completed BDS for response data
	<code>bds_buffptr</code>	
	<code>bds_buffsize</code>	

Output

<--	<code>atd_status</code>	error code
<--	<code>atd_tranid</code>	transaction ID for this request
<--	<code>atp_bdsresps</code>	number of valid BDSElements at <code>atp_bdsptr</code>
<--	<code>atp_bdsptr</code>	data information from response
	<code>bds_datasize</code>	
	<code>bds_usrbytes</code>	

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NO_MEM_ERROR
ATP_REQFAILED
ATP_OVERFLOW
ATP_CANCELLED

Figure 8-5 illustrates the parameter block for the ATPSendRequest command.

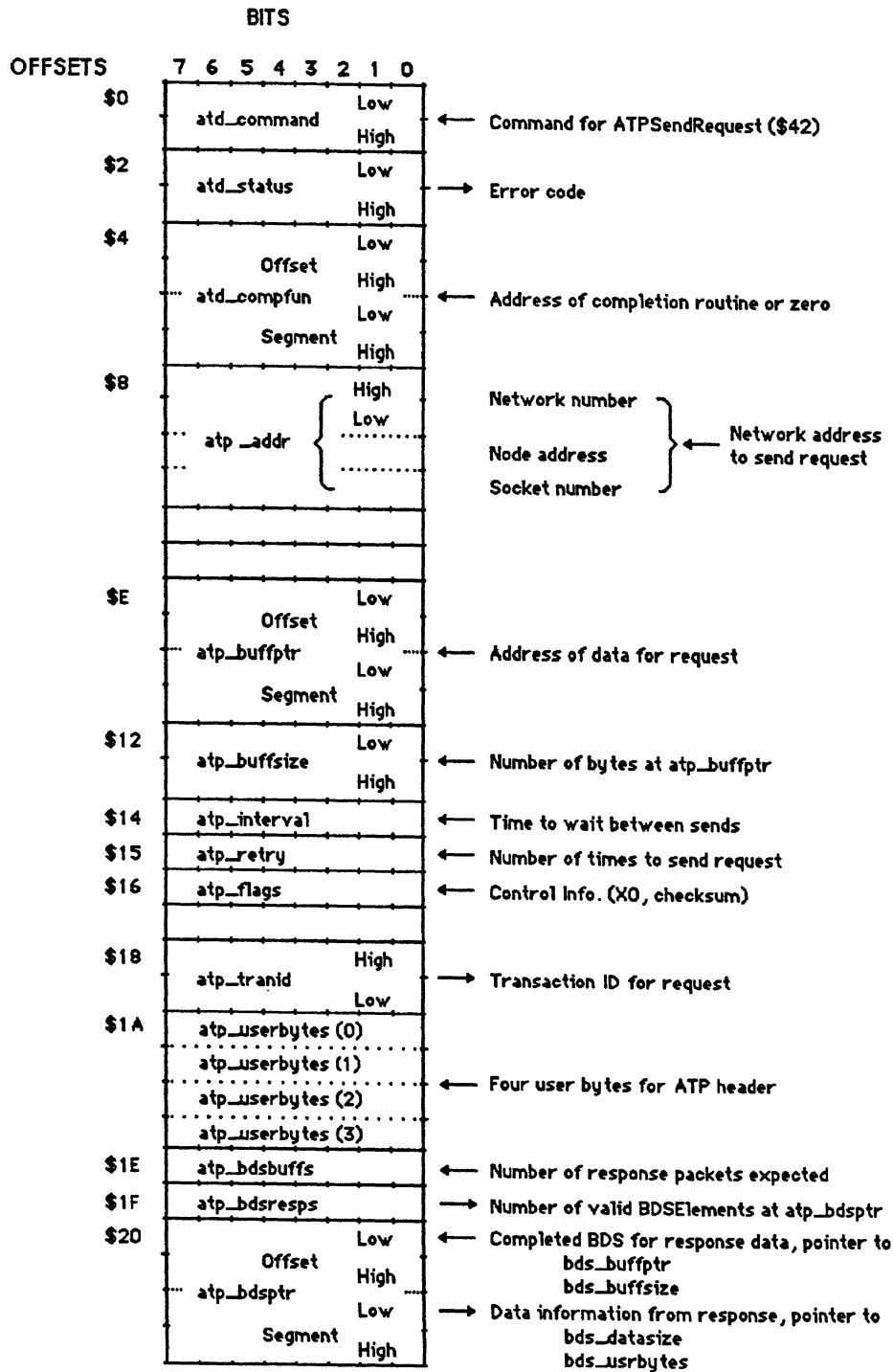


Figure 8-5. ATPSendRequest Parameter Block

ATPGetRequest (\$43)

The ATPGetRequest command sets up the mechanism to receive a request initiated on another node by an ATPSendRequest call.

The atp_socket field indicates at which socket the request should be “listened” for (returned by the ATPOpenSocket call).

The fields atp_buffptr and atp_buffsize fields indicate where the incoming request should be placed and the size of that buffer.

If nonzero, the atp_compfun field should contain the address of code to be executed when a request is received. The retry and interval information is not used.

Other information about the incoming request is placed in the parameter structure after the request is received: the transaction ID is placed in the atp_tranid field, the bit map field of the ATP header goes in the atp_seqbit field, the ATP control flags go in the atp_flags field, and the source of the request is placed in the atp_addr field. The atp_buffsize field is updated to reflect the actual size of the incoming request data. If the incoming request had a DDP checksum, the CHKSUM bit of the atp_flag field will be set.

Parameter usage and error returns for the ATPGetRequest command are as follows:

Parameter Usage

Input

-->	atd_command	AtpGetRequest
-->	atd_compfun	address of completion routine or 0
-->	atp_socket	which socket to listen on
-->	atp_buffptr	segment:offset of buffer for incoming request data
-->	atp_buffsize	size of buffer at atp_buffptr

Output

<--	atd_status	error code
<--	atp_addr	network address of requester
<--	atp_buffsize	number of bytes at atp_buffptr
<--	atp_flags	control flags from ATP header
<--	atp_seqbit	bitmap from ATP header
<--	atp_tranid	ID of this transaction
<--	atp_userbytes	four user bytes from ATP header

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NO_MEM_ERROR
ATP_BADSOCKET
ATP_OVERFLOW
ATP_CANCELLED

Figure 8-6 illustrates the parameter block for the ATPGetRequest command.

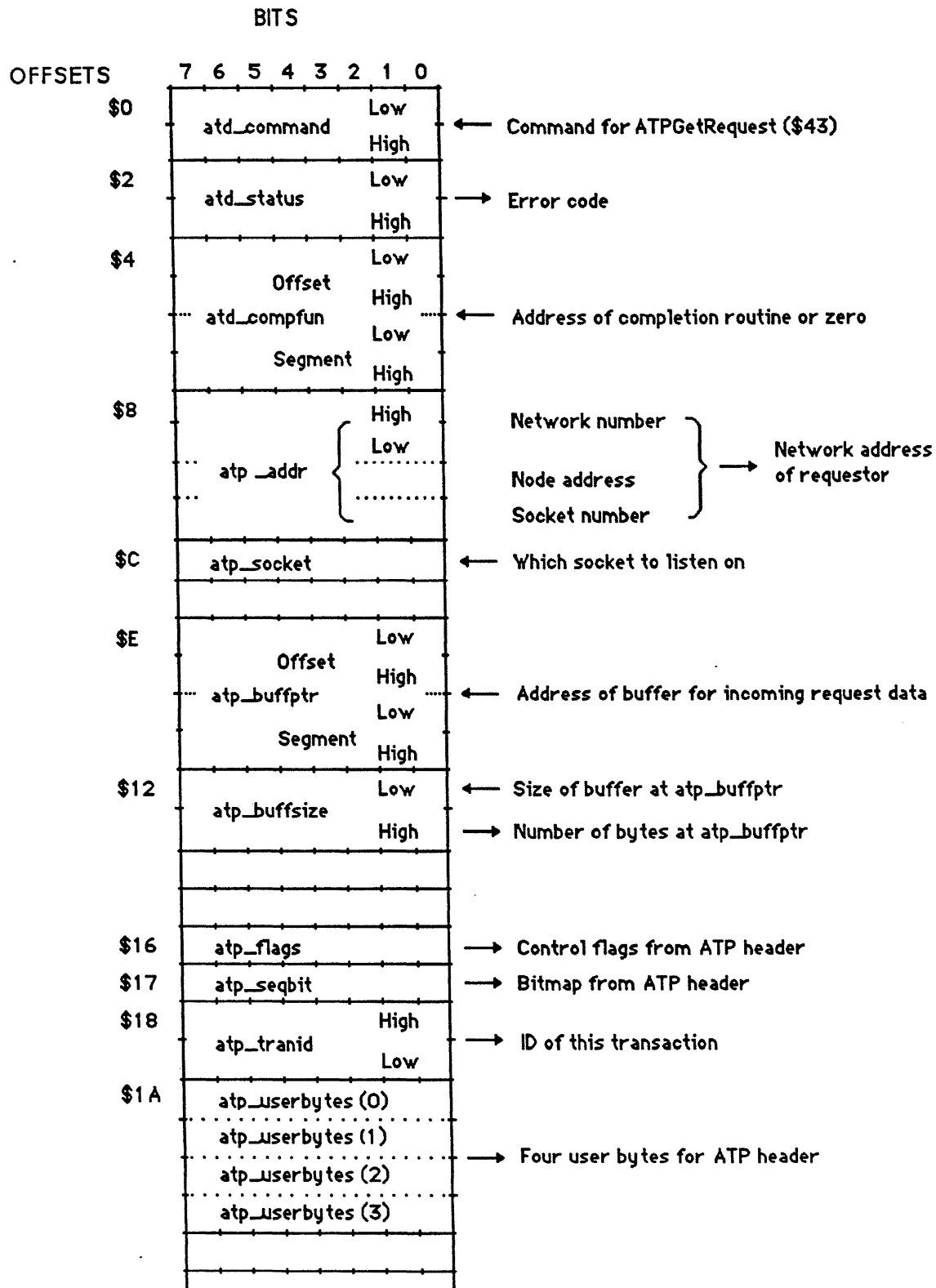


Figure 8-6. ATPGetRequest Parameter Block

ATPSendResponse (\$44)

The ATPSendResponse command sends from zero to eight response packets as a full or partial response to a request. (More responses can be added with the ATPAddResponse command, described later in this chapter.)

The embedded AddrBlk contains the socket to which the packets should be sent, and the atp_socket field is the socket they should be sent through.

The BDS at the atp_bdspr field has atp_bdsbuffs elements, and atp_bdsresps of those elements are currently filled in with valid response information and should not be sent out. Any elements more than the number of current responses may be used with subsequent ATPAddResponse calls.

The atp_flags field should have the “end-of-response” bit (EOMbit value is 10H) set if this call will send all response packets for the transaction *and*, the number of response packets being sent is less than the number expected (according to the atp_seqbit bitmask sent with the original request). The atp_flags field should have the CHECKSUM bit set if the caller wants checksums to be computed on Internet response packets. Setting the STSbit in the atp_flags field will result in that bit being set in the last response packet that is sent as part of this command. The atp_tranid field has the transaction number returned by ATPGetRequest.

If an ATPSendResponse command to an exactly once transaction is going to be followed with one or more ATPAddResponse commands, the parameter block must be maintained by the caller until all of the ATPAddResponses calls and the ATPSendResponses calls have completed. The caller may also make use of the bitmask in the atp_seqbit field, as ATP keeps the current status of the responses here. If there is a completion function for this command, for exactly once transactions, the completion function will not be called until the release has been received; thus, it may not be called until ATPAddResponse calls are made.

Note: One of the error codes for this command is ATP_NORELEASE, which means that the exactly once requestor does not send a release packet that indicates all response data has been received. This error code does not necessarily mean there has been an error in completing of the transaction.

Parameter usage and error returns for the ATPSendResponse command are listed on the following page.

Parameter Usage

Input

--> atd_command	ATPSendResponse
--> atd_compfun	address of completion routine or 0
--> atp_addr	where to send response packets
--> atp_socket	which socket to send through
--> atp_flags	EOMbit set if end of responses, CHECKSUM bit & STSbit may be set
--> atp_tranid	ID of this transaction
--> atp_bdsbuffs	number BDSElements at atp_bdsptr
--> atp_bdsresps	number of currently valid BDSElements at atp_bdsptr
--> atp_bdsptr	data information for responses
	bds_buffptr
	bds_datasize
	bds_usrbytes

Output

<-- atd_status	error code
<-- atp_seqbit	bitmap from request resend

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NO_MEM_ERROR
ATP_BADSOCKET
ATP_NOSENDRESPONSE
ATP_NORELEASE
ATP_CANCELLED

Figure 8-7 illustrates the parameter block for the ATPSendResponse command.

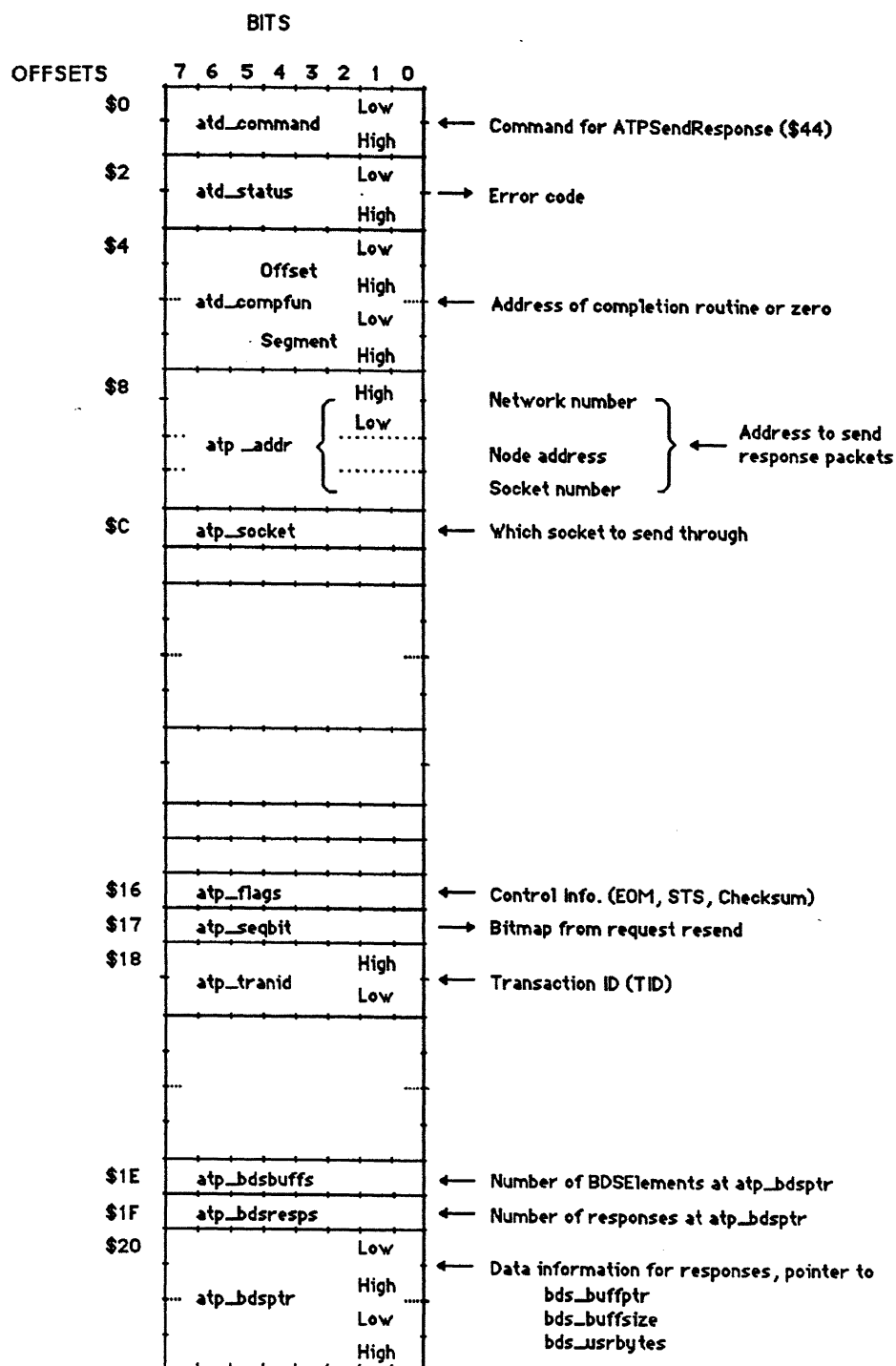


Figure 8-7. ATPSendResponse Parameter Block

ATPAddResponse (\$45)

The ATPAddResponse command is used to send one response packet in conjunction with a previously executed asynchronous ATPSendResponse.

Although it must be called with a different parameter block than the original, the following fields should contain the same values: atp_socket, atp_addr, and atp_tranid. In addition to these fields, the atp_flags field contains control information (EOM, for example), and the atp_seqbit field should have the sequence number of this response. The atp_flags field should have the CHECKSUM bit set if the caller wants checksums to be computed on Internet response packets. Setting the STSbit in the atp_flags field will result in that bit being set in the response packet that is sent as part of this command.

A completion routine may be passed in the atp_compfun field, but it should be noted that the completion of this call does not necessarily mean that the transaction has been completed. No BDS information is required, but the response data to be sent is addressed by the atp_buffptr field, and atp_buffsize bytes will be sent.

The atp_userbytes field in the parameter structure should hold the four user bytes that go in the ATP header. This information, which usually goes in a BDSElement, is taken from the ATPParams structure and is placed in the BDS of the original ATPSendResponse.

Parameter usage and error returns for the ATPAddResponse command are as follows:

Parameter Usage

Input

--> atd_command	ATPAddResponse
--> atd_compfun	address of completion routine or 0
--> atp_addr	network address to send request
--> atp_socket	socket to send through
--> atp_buffptr	segment:offset of data for response
--> atp_buffsize	number of bytes at atp_buffptr
--> atp_flags	EOM bit set for end-of-message, CHECKSUM bit & STSbit may be set
--> atp_seqbit	sequence number of this response
--> atp_tranid	ID of transaction
--> atp_userbytes	four user bytes for ATP header

Output

<-- atd_status	error code
----------------	------------

Error Returns

```

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NO_MEM_ERROR
ATP_BADSOCKET
ATP_NOSENDRESPONSE
ATP_CANCELLED
    
```

Figure 8-8 illustrates the parameter block for the ATPAddResponse command.

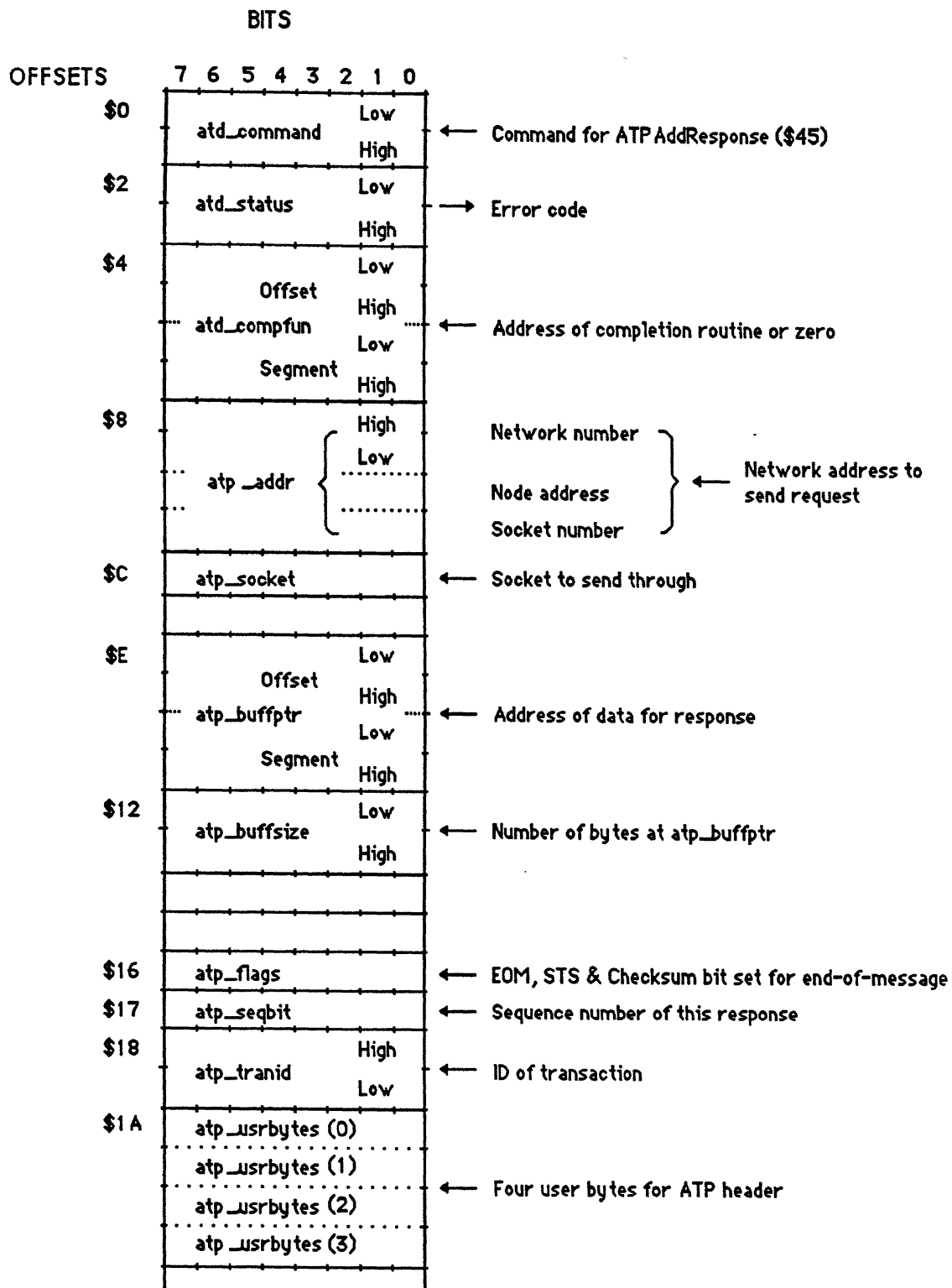


Figure 8-8. ATPAddResponse Parameter Block

ATPCancelTrans (\$46)

The ATPCancelTrans command cancels any outstanding transaction.

The parameter structure address that is passed in the `atp_buffptr` field must be the same parameter structure as the transaction being cancelled. If the command being cancelled had a completion routine associated with it, that routine will be called with the `atp_status` field set to `ATP_CANCELLED`.

Note: Using the ATPCancelTrans command is the only way to cancel an outstanding ATPGetRequest call.

Parameter usage and error returns for the ATPCancelTrans command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	ATPCancelTrans
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>atp_buffptr</code>	params of command being cancelled

Output

<--	<code>atd_status</code>	error code
-----	-------------------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER

Figure 8-9 illustrates the parameter block for the ATPCancelTrans command.

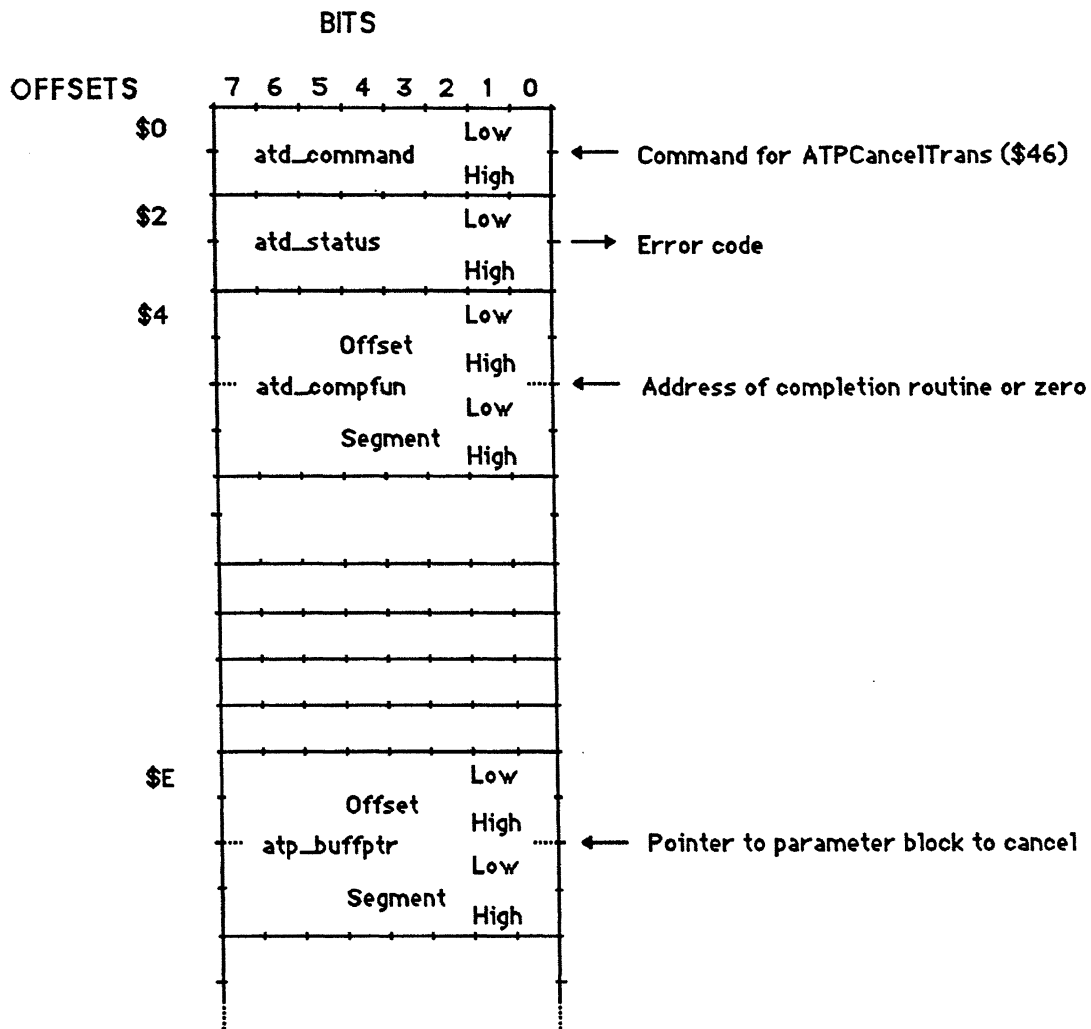


Figure 8-9. ATPCancelTrans Parameter Block

ATPCancelResponse (\$47)

The ATPCancelResponse command cancels an outstanding ATPSendResponse and will free the Driver's internal "Response Control Block" that is used for filtering duplicate exactly once requests on a completed ATPGetRequest.

In both cases, the atp_addr field should be set to the Request source address, the atp_socket field should be set to the transaction's socket, and the atp_tranid field should be set to the appropriate transaction ID. If the command being cancelled had a completion routine associated with it, that routine will be called with the atd_status field set to ATP_CANCELLED. This command is especially useful when the NO_RELEASE bit is being used in ATP_flags.

Parameter usage and error returns for the ATPCancelResponse command are as follows:

Parameter Usage

Input

--> atd_command	ATPCancelResponse
--> atd_compfun	address of completion routine or 0
--> atp_addr	source address or filter AddrBlk
--> atp_socket	socket used for GetRequest
--> atp_tranid	transaction ID

Output

<-- atd_status	error code
----------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER

Figure 8-10 illustrates the parameter block for the ATPCancelResponse command.

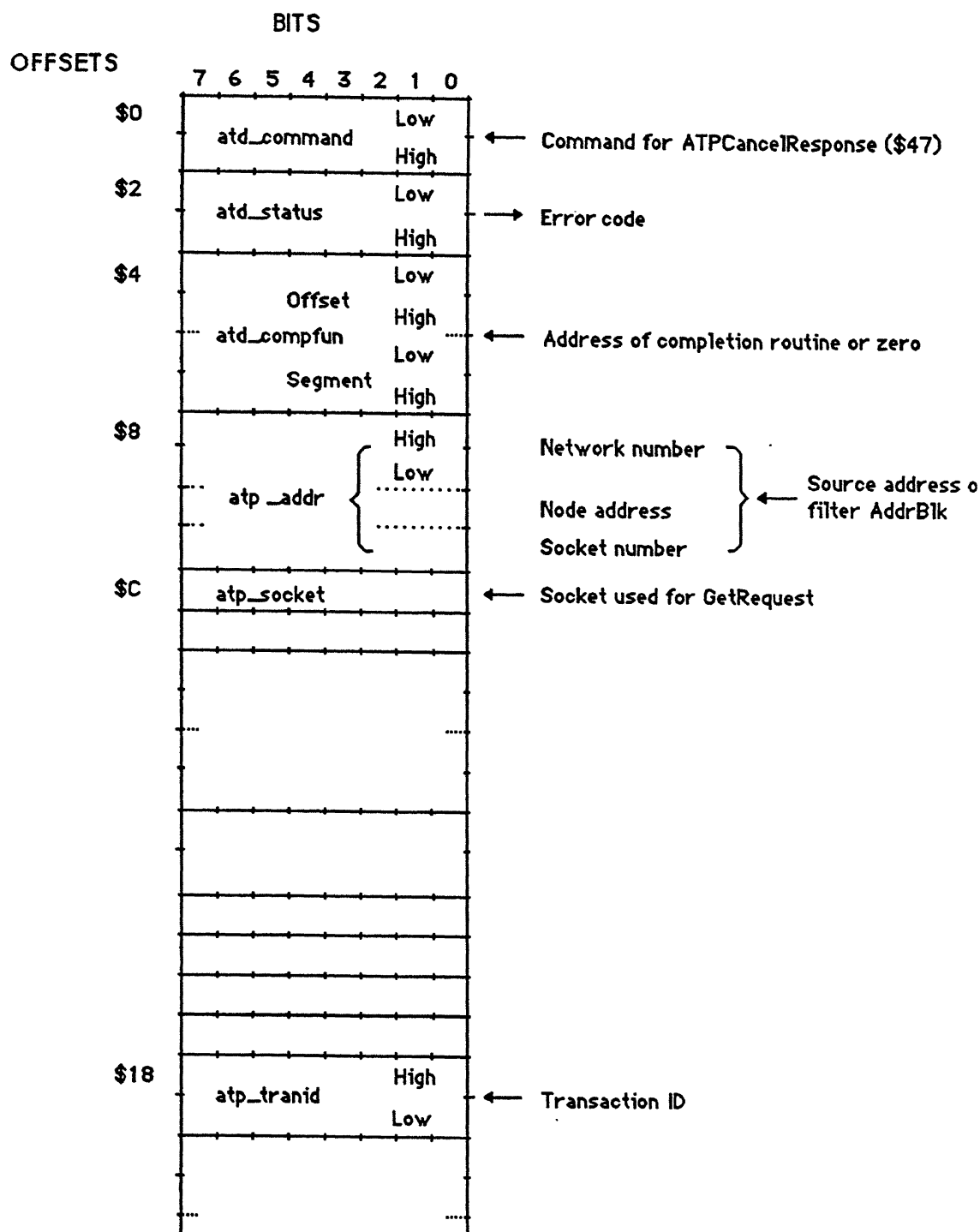


Figure 8-10. ATPCancelResponse Parameter Block

ATPCancelRequest (\$48)

The ATPCancelRequest command cancels an outstanding ATPSendRequest.

The passed parameter structure should have the `atp_addr` and `atp_tranid` fields set to the same values as the transaction being cancelled. If the ATPSendRequest being cancelled had a completion routine associated with it, that routine will be called with the `atd_status` field set to `ATP_CANCELLED`. This command is especially useful when the `NO_RELEASE` bit is being used in `ATP_flags`.

Parameter usage and error returns for the ATPCancelRequest command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	ATPCancelRequest
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>atp_addr</code>	destination address of request
-->	<code>atp_tranid</code>	transaction ID

Output

<--	<code>atd_status</code>	error code
-----	-------------------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER

Figure 8-11 illustrates the parameter block for the ATPCancelRequest command.

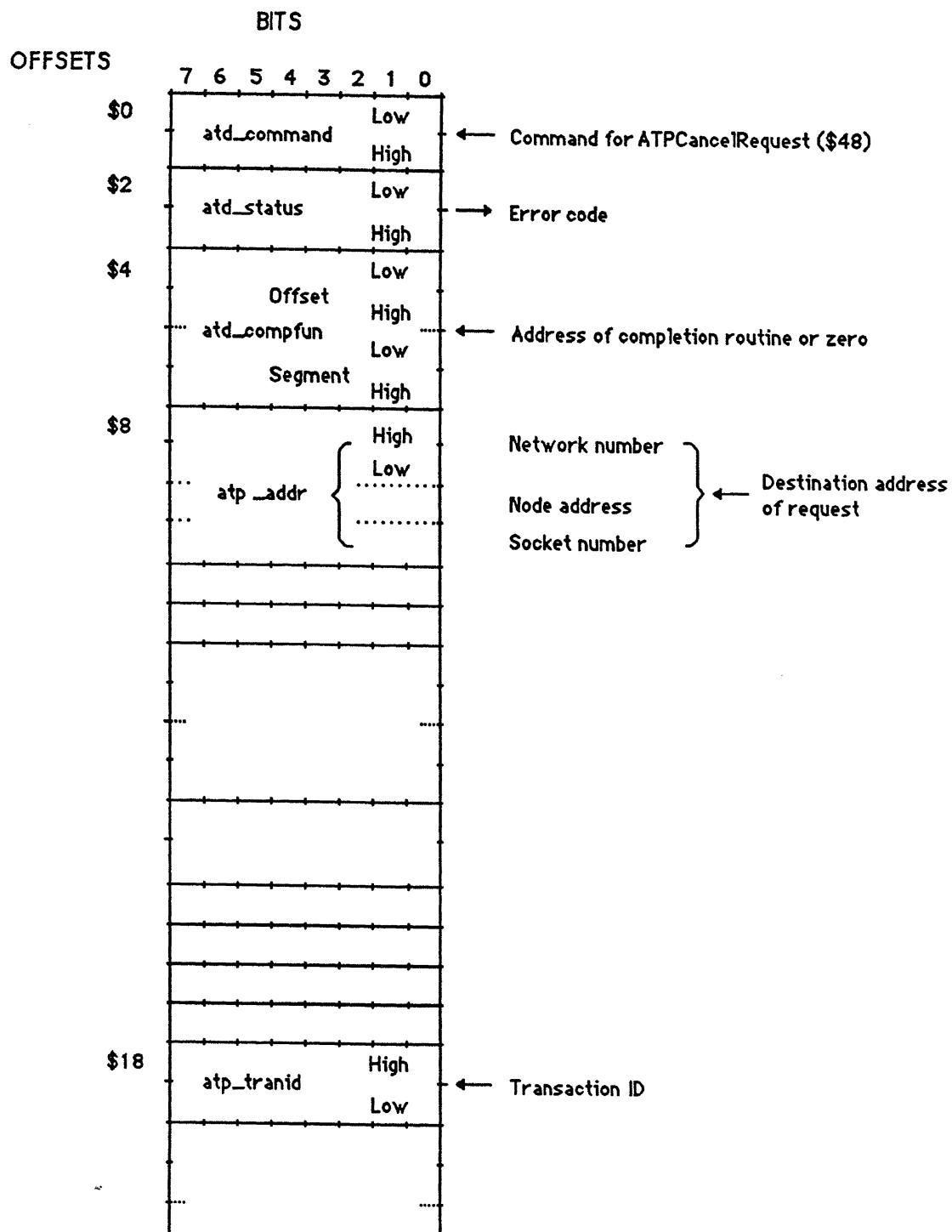


Figure 8-11. ATPCancelRequest Parameter Block

ATPRespond (\$49)

The ATPRespond command sends from zero to eight response packets as a full (not partial) response to a request.

The embedded AddrBlk contains the address to which the packets should be sent, and the atp_socket field is the socket they should be sent through.

The atp_flags field should have the end-of-response bit (EOMbit value is 10H) set if the number of response packets being sent is less than the number expected (according to the atp_seqbit bit mask sent with the original request). The atp_flags field should have the CHECKSUM bit set if the caller wants checksums to be computed on Internet response packets. Setting the STSbit in the atp_flags field will result in that bit being set in the last response packet that is sent as part of this command. The atp_tranid field has the transaction number returned by ATPGetRequest.

Some protocol (notably PAP) require that the number of bytes per response transaction be less than 578. The offset of the BDSptr in the parameter block contains the quanta to break up the response buffer into (PAP uses 512). Most other protocol will use 578 (a zero byte here implies 578).

If the ATPRespond command is an exactly once transaction, the parameter block must be maintained by the caller until the ATPRespond is completed. The caller may also make use of the bit mask in the atp_seqbit field, as ATP keeps the current status of the responses here. If there is a completion function for this command, for exactly once transactions, the completion function will not be called until the release has been received.

Note: One of the error codes for this command is ATP_NORELEASE, which means that the exactly once requestor does not send a release packet to indicate all response data has been received. This error code does not necessarily mean there has been an error in completing of the transaction.

The user bytes sent in the response will be those stored in atp_userbytes and will be sent in ALL the response packets.

Parameter usage and error returns for the ATPRespond command are listed on the following page.

AppleTalk PC Card and Driver Preliminary Note

Parameter Usage

Input

--> atd_command	ATPRespond
--> atd_compfun	address of completion routine or 0
--> atp_addr	where to send response packets
--> atp_socket	which socket to send through
--> atp_buffptr	segment:offset of response data buffer
--> atp_buffsize	amount of data at above address
--> atp_flags	EOMbit set if end of responses, CHECKSUM bit & STSbit may be set
--> atp_tranid	ID of this transaction
--> atp_userbytes	four user bytes for ATP header
--> atp_bdsptr	Quanta size, chunk size 0 = 578 - in offset of this pointer - determines how big each BDS transaction will be
--> atp_userbytes	the user bytes to include in each element of the response

Output

<-- atd_status	error code
<-- atp_seqbit	bit map from request resent

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NO_MEM_ERROR
ATP_BADSOCKET
ATP_NOSENDRESPONSE
ATP_NORELEASE
ATP_CANCELLED

Figure 8-12 illustrates the parameter block for the ATPRespond command.

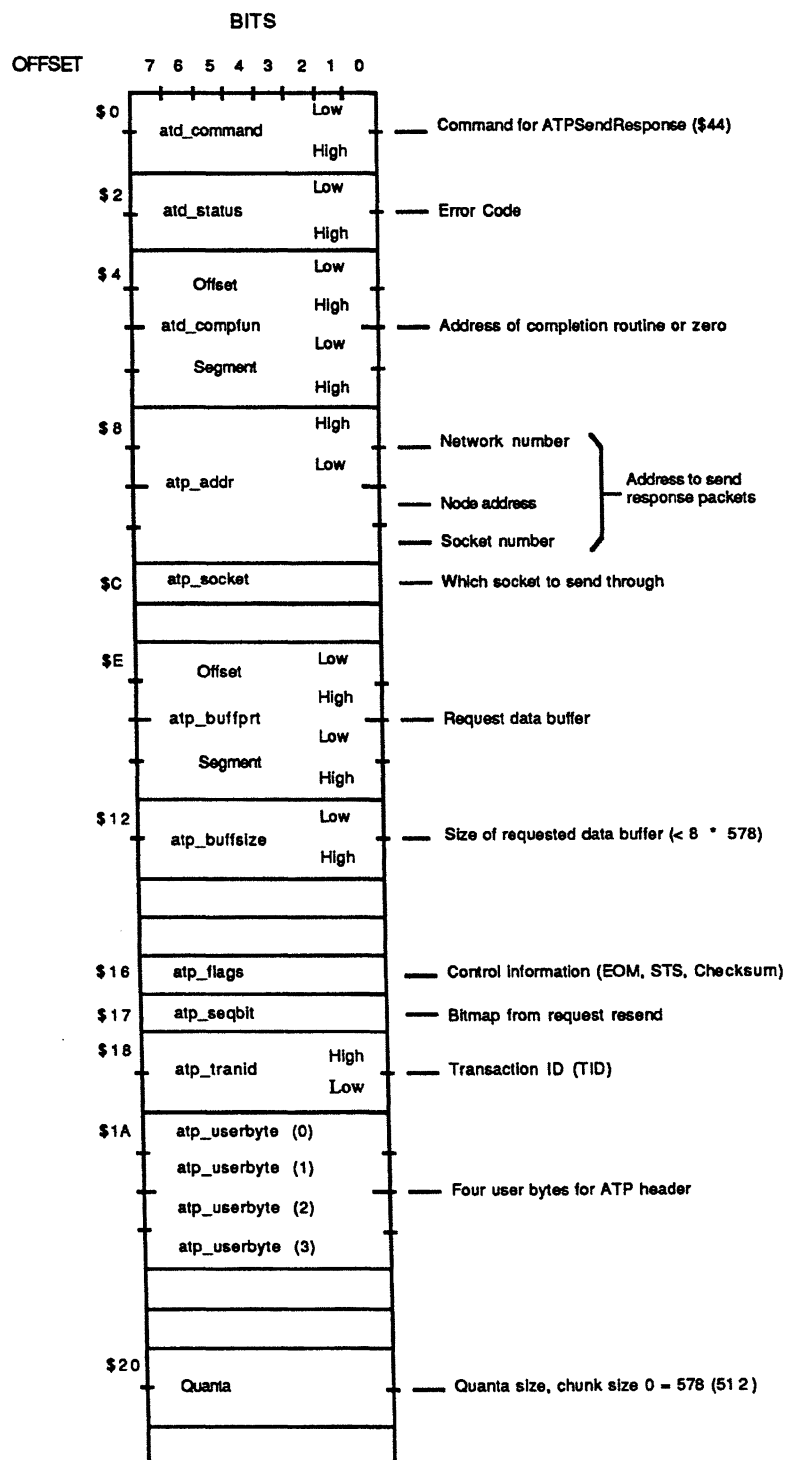


Figure 8-12. ATPRespond Parameter Block

Chapter 9

Printer Access Protocol (PAP) Commands

Introduction

The Printer Access Protocol (PAP) commands create and maintain a connection between a workstation and a server (a printer).

This chapter describes the interface to the workstation-related PAP calls. It is possible for a driver implementation to support only one group of these commands, as indicated by the configuration bit map in `ATGetNetInfo`. The commands included with the PAP Workstation (PAPWks) protocol groups are as follows:

- PAPRead
- PAPWrite
- PAPCancel

- PAPOpen
- PAPStatus
- PAPClose

In all cases, a return value of zero means that no errors have yet occurred, and a negative number signifies some error. There may be only one session open per server.

For commands made asynchronously, it is important that the caller not modify the parameter blocks passed to PAP until the command has completed. The application may monitor the status word of the passed parameter structure for completion. Until the call is complete, this status word will have a positive value. Upon completion, that word will be zero if there were no errors, or a negative number if there was an error.

Protocol Parameter Structures

The following structures are used in the interface to the PAP commands. The first structure illustrates the form of a "status record" returned by a printer entity that is pointed to by the `pap_bptr` field.

```
PAPStatusRec
    dword    psr_system;    internal PAP information
    char[256] psr_status;    printer-dependent status string
```

Figure 9-1 illustrates the parameter block of a status record:

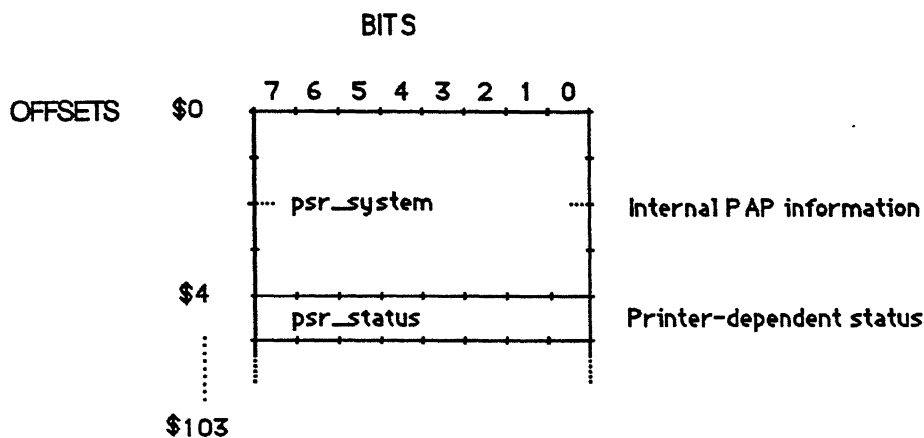


Figure 9-1. Parameter Block for Status Record

The second structure describes the general PAP parameter structure:

```
PAPParams
    word    atd_command;    command code
    word    atd_status;    status word/error return
    dword   atd_compfun;    completion routine or zero
    AddrBlk pap_addr;    address for status, open calls
    word    pap_refnum;    reference number for connection
    dword   pap_buffptr;    segment:offset of buffer area
    word    pap_buffsize;    size of buffer area
    byte    pap_eof;    end-of-file indicator
    byte    pap_srefnum;    server reference number
    dword   pap_entptr;    segment:offset of EntityName
```

Figure 9-2 illustrates the general PAP parameter structure.

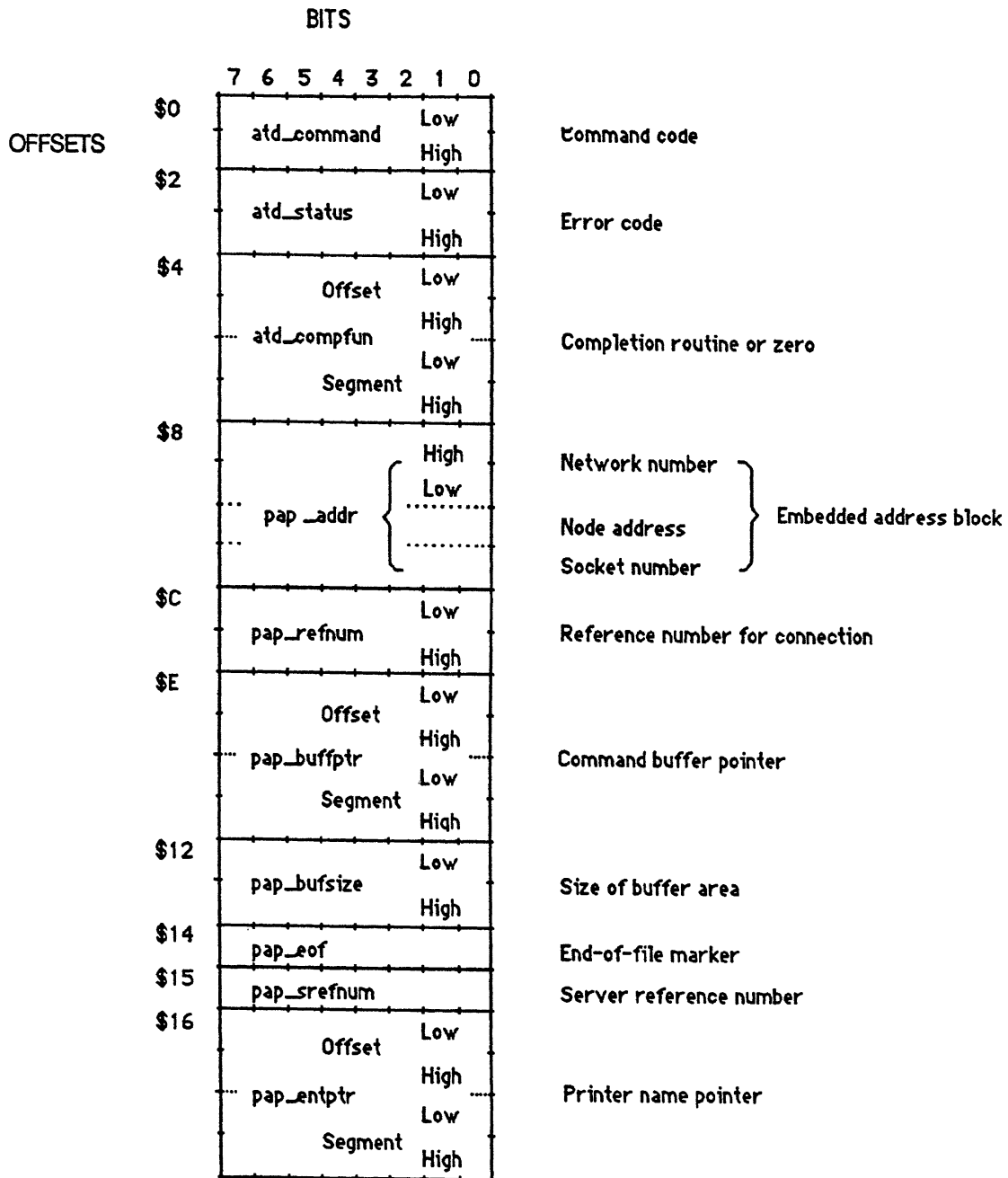


Figure 9-2. General Parameter Block for Printer Access Protocol (PAP) Commands

PAP Commands for the Workstation

This section describes and illustrates the PAP commands common to a workstation:

- PAPRead
- PAPWrite
- PAPCancel

PAPRead (\$72)

The PAPRead command is made by a workstation to read data on a PAP connection (sent by the other end of the connection by using a PAPWrite).

The Parameter fields required as input are `pap_refnum`, which holds the connection reference number, and `pap_buffptr`, which gives the address of the buffer where the incoming data should be placed. It is assumed that the buffer is at least as big as the flow quantum (multiplied by 512) specified in the PAPOpen call. If it is not, results may be unpredictable.

Upon successful completion, the `pap_buffsize` field indicates how many bytes of data were placed at the `pap_buffptr` field, and the `pap_eof` field is non-zero if an end-of-file condition is received from the other end of the connection.

Parameter usage and error returns for the PAPRead command are as follows:

Parameter Usage

Input

-->	<code>atd_command</code>	PAPRead
-->	<code>atd_compfun</code>	address of completion routine or 0
-->	<code>pap_refnum</code>	reference number for connection
-->	<code>pap_buffptr</code>	segment:offset of read buffer
-->	<code>pap_buffsize</code>	number of bytes to read

Output

<--	<code>atd_status</code>	error code
<--	<code>pap_buffsize</code>	number of bytes read
<--	<code>pap_eof</code>	set if end-of-file

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
PAP_BADCONNID
PAP_DIED
PAP_LENERR

Figure 9-3 illustrates the parameter block for the PAPRead command.

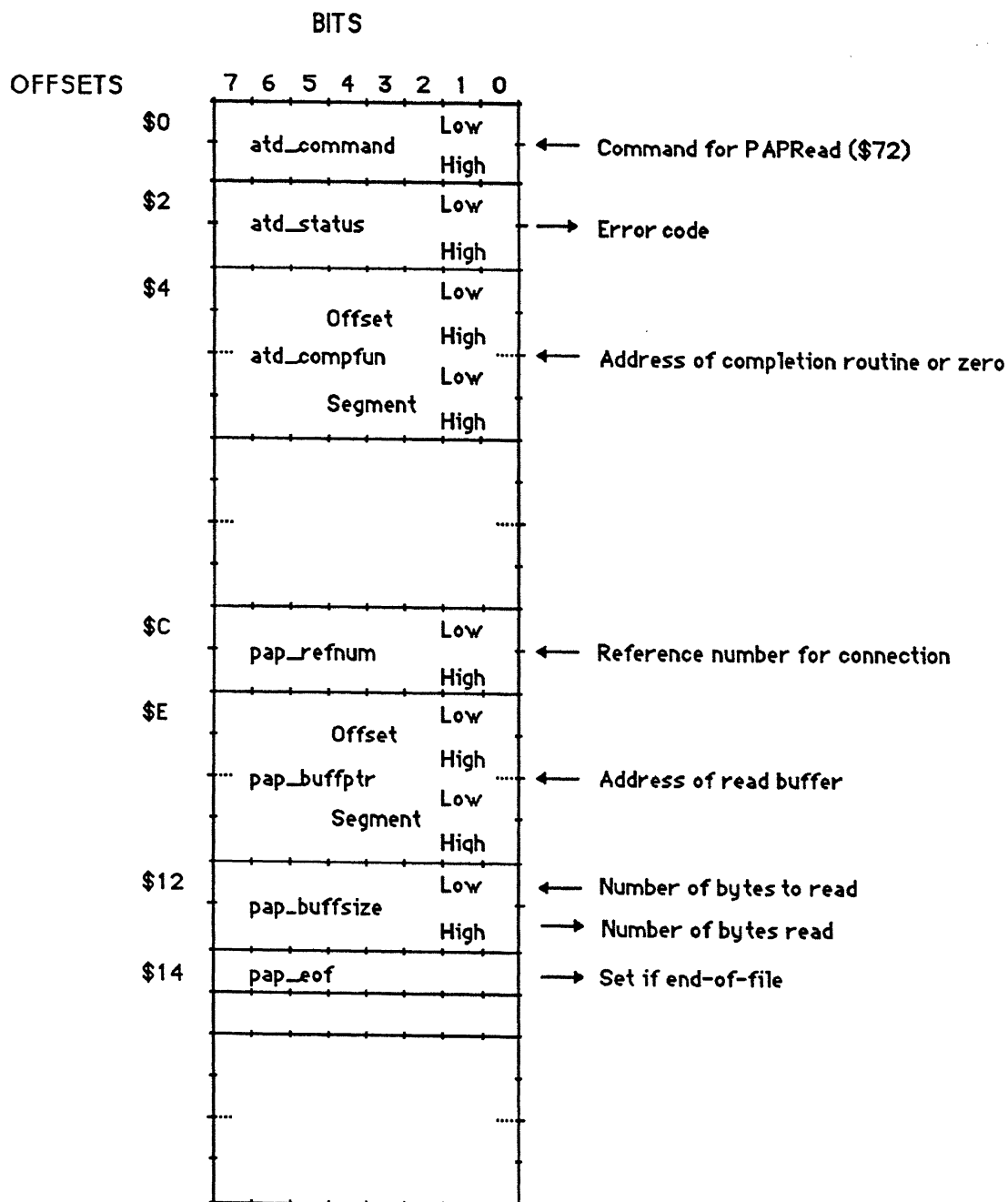


Figure 9-3. PAPRead Parameter Block

PAPWrite (\$73)

The PAPWrite command is made by a workstation to write data on a PAP connection (read by the other end of the connection by using a PAPRead).

The parameter fields required as input are pap_refnum, which holds the connection reference number, and pap_buffptr, giving the address of the buffer from which the data should be sent. The pap_buffsize field indicates how much data (in bytes) should be sent, and the pap_eof field should be non-zero if an end-of-file indicator should be sent to the other end of the connection.

Parameter usage and error returns for the PAPWrite command are as follows:

Parameter Usage

Input

-->	atd_command	PAPWrite
-->	atd_compfun	address of completion routine or 0
-->	pap_refnum	reference number for connection
-->	pap_buffptr	segment:offset of write data
-->	pap_buffsize	number of bytes to be sent
-->	pap_eof	set if end-of-file

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
PAP_BADCONNID
PAP_DIED
PAP_LENERR

Figure 9-4 illustrates the parameter block for the PAPWrite command.

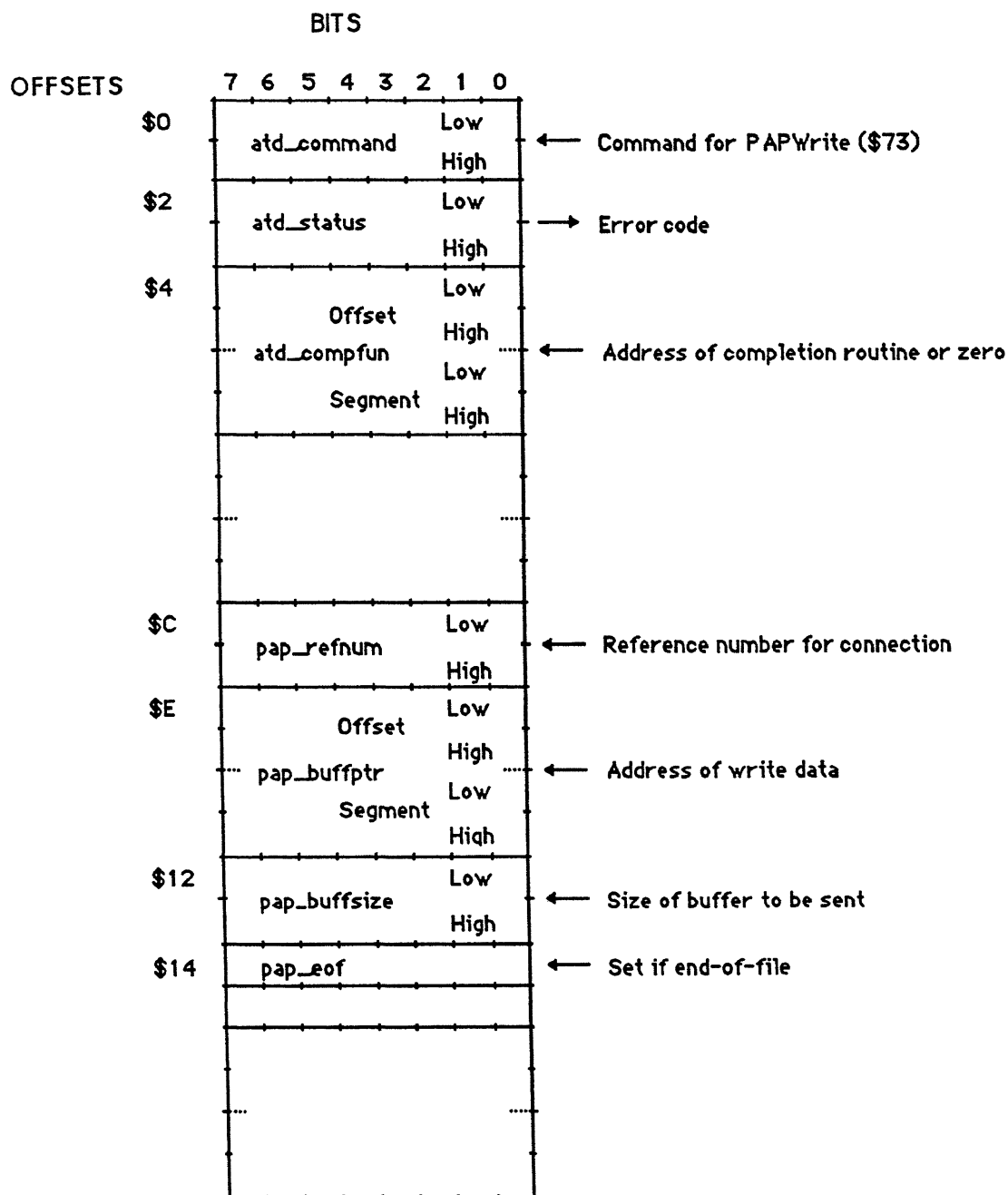


Figure 9-4. PAPWrite Parameter Block

PAPCancel (\$7B)

The PAPCancel command cancels an outstanding PAP command. The pap_buffptr field should contain the segment:offset of the parameter block passed in with the command being cancelled. Because PAP is a sequenced session protocol, the cancellation of some outstanding commands could effectively break the session. Therefore, only the PAPOpen and PAPStatus commands may be cancelled with PAPCancel.

Parameter usage and error returns for the PAPCancel command are as follows:

Parameter Usage

Input

-->	atd_command	PAPCancel
-->	atd_compfun	address of completion routine or 0
-->	pap_buffptr	parameter block of command to cancel

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
CALLNOTSUPPORTED

Figure 9-5 illustrates the parameter block for the PAPCancel command.

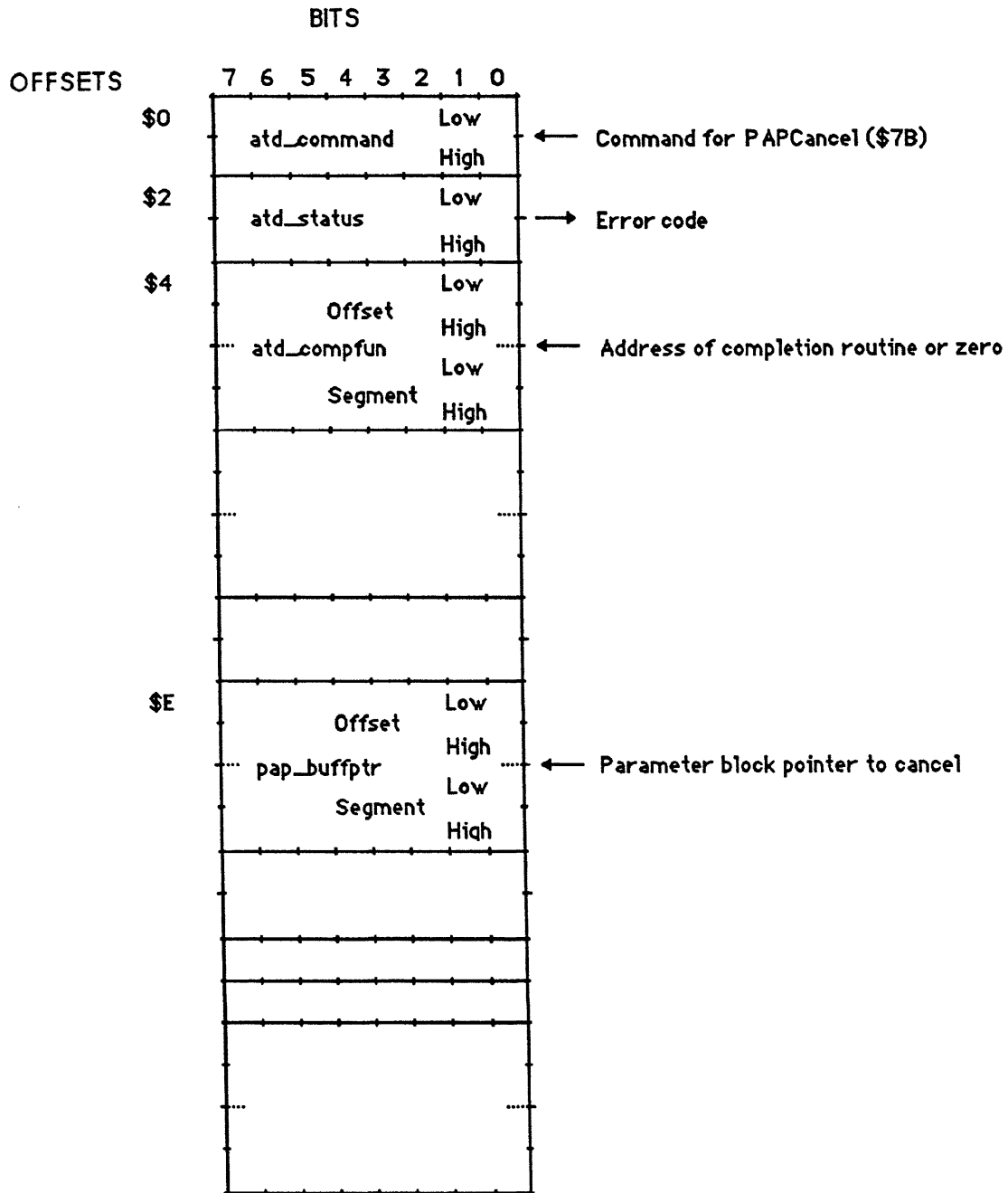


Figure 9-5. PAPCancel Parameter Block

PAP Workstation Commands

This section describes the PAP commands for the workstation:

- PAPOpen
- PAPClose
- PAPStatus

PAPOpen (\$70)

The PAPOpen command is made by a workstation to open a connection with a printer entity (or any PAP server) on the network. On this connection, the workstation may send data to the printer (via PAPWrite) or read data from the printer (via PAPRead).

The required inputs are a valid network address in the pap_addr field or a pap_addr field of zero and a pointer to the printer entity's name (in the form of an EntityName), which is passed in the pap_entptr field. Also required are a flow quantum for PAPReads, which is passed in the pap_eof field, and a buffer for placing the status returned by the printer entity, which is passed in the pap_buffptr field.

Upon successful completion, the pap_refnum field of the parameter structure will contain a valid reference number to be used with the connection. The status record (as described earlier in "Protocol Parameter Structures" in this chapter) will have the printer-dependent status information. If not specified as an input, the pap_addr field will have the network address of the printer entity for PAPStatus calls.

Parameter usage and error returns for the PAPOpen command are as follows:

Parameter Usage

Input

-->	atd_command	PAPOpen
-->	atd_compfun	address of completion routine or 0
-->	pap_addr	network address of printer or 0
-->	pap_buffptr	segment:offset of PAPStatusRec
-->	pap_eof	quantum value (for PAPReads)
-->	pap_entptr	segment:offset of EntityName

Output

<--	atd_status	error code
<--	pap_addr	network address of PAP server
<--	pap_refnum	reference number for connection
<--	pap_eof	printer's quantum value (for PAPReads)

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
PAP_NOCONNIDS
NBP_NOTFOUND
NO_MEM_ERROR

Figure 9-6 illustrates the parameter block for the PAPOpen command.

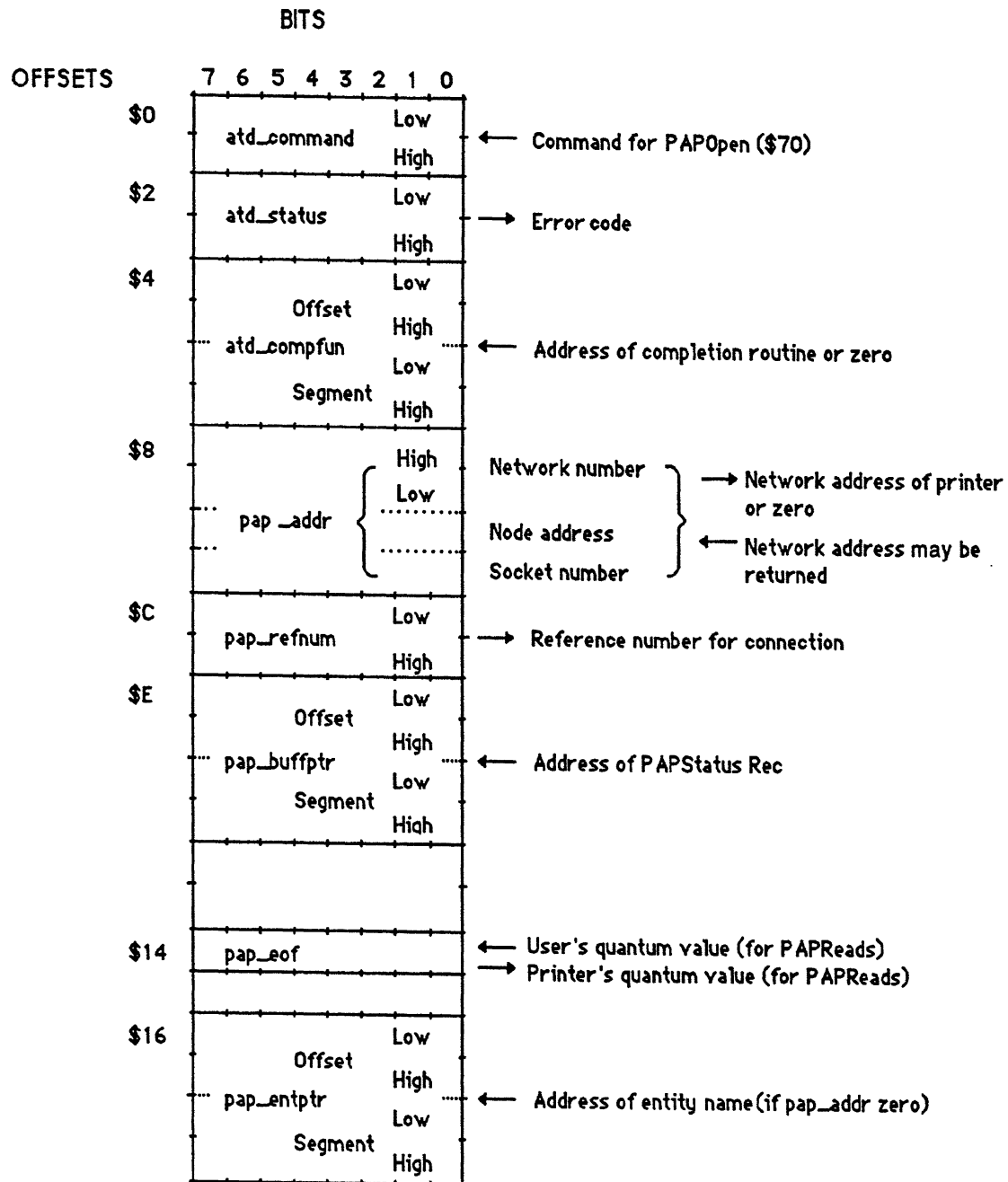


Figure 9-6. PAPOpen Parameter Block

PAPClose (\$71)

The PAPClose command is made by a workstation to close an open connection with a printer entity. The input is the reference number, in the form of RefNum, returned with a PAPOpen.

Parameter usage and error returns for the PAPClose command are as follows:

Parameter Usage

Input

-->	atd_command	PAPClose
-->	atd_compfun	address of completion routine or 0
-->	pap_refnum	reference number for connection

Output

<--	atd_status	error code
-----	------------	------------

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
PAP_BADCONNID
PAP_DIED

Figure 9-7 illustrates the parameter block for the PAPClose command.

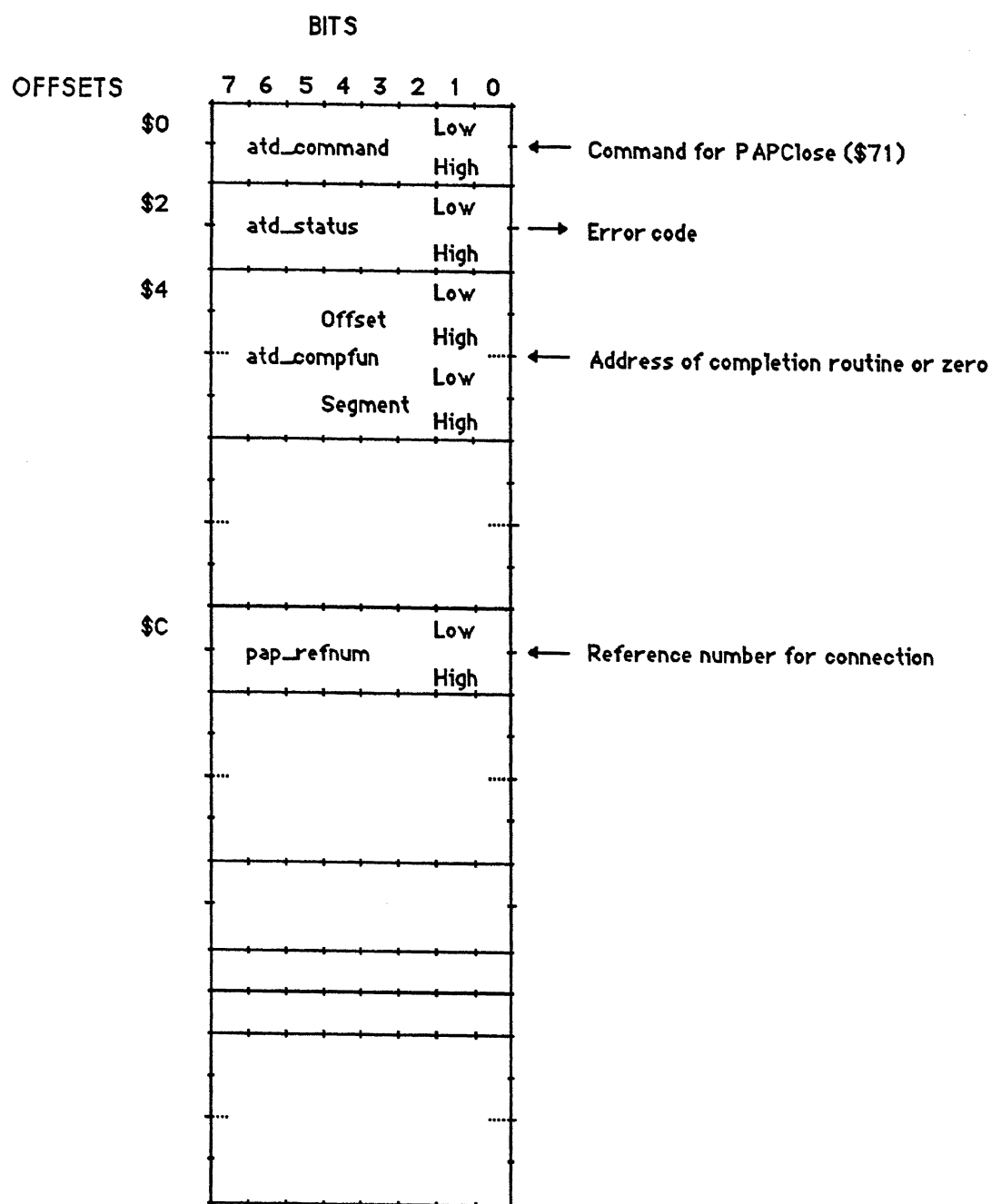


Figure 9-7. PAPClose Parameter Block

PAPStatus (\$74)

The PAPStatus call is made by a workstation to obtain the current status of a printer entity on the network.

The required inputs are a valid network address in the pap_addrfield or a pap_addr field of zero and a pointer to the printer entity's name (in the form of an EntityName), which is passed in the pap_entptr field. Also required is a buffer for placing the status returned by the printer entity, which is passed in the pap_buffptr field.

Upon successful completion, the status record at the pap_buffptr field (as described earlier in "Protocol Parameter Structures" in this chapter) will have the current printer-dependent status information and the pap_addr field, if zero on input, will be filled in by the driver.

Implied retry and interval are 5 and 2, respectively.

Parameter usage and error returns for the PAPStatus command are as follows:

Parameter Usage

Input

-->	atd_command	PAPStatus
-->	atd_compfun	address of completion routine or 0
-->	pap_addr	entity address or zero
-->	pap_buffptr	segment:offset of PAPStatusRec
-->	pap_entptr	segment:offset of EntityName

Output

<--	atd_status	error code
<--	pap_addr	entity address

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NBP_NOTFOUND

Figure 9-8 illustrates the parameter block for the PAPStatus command.

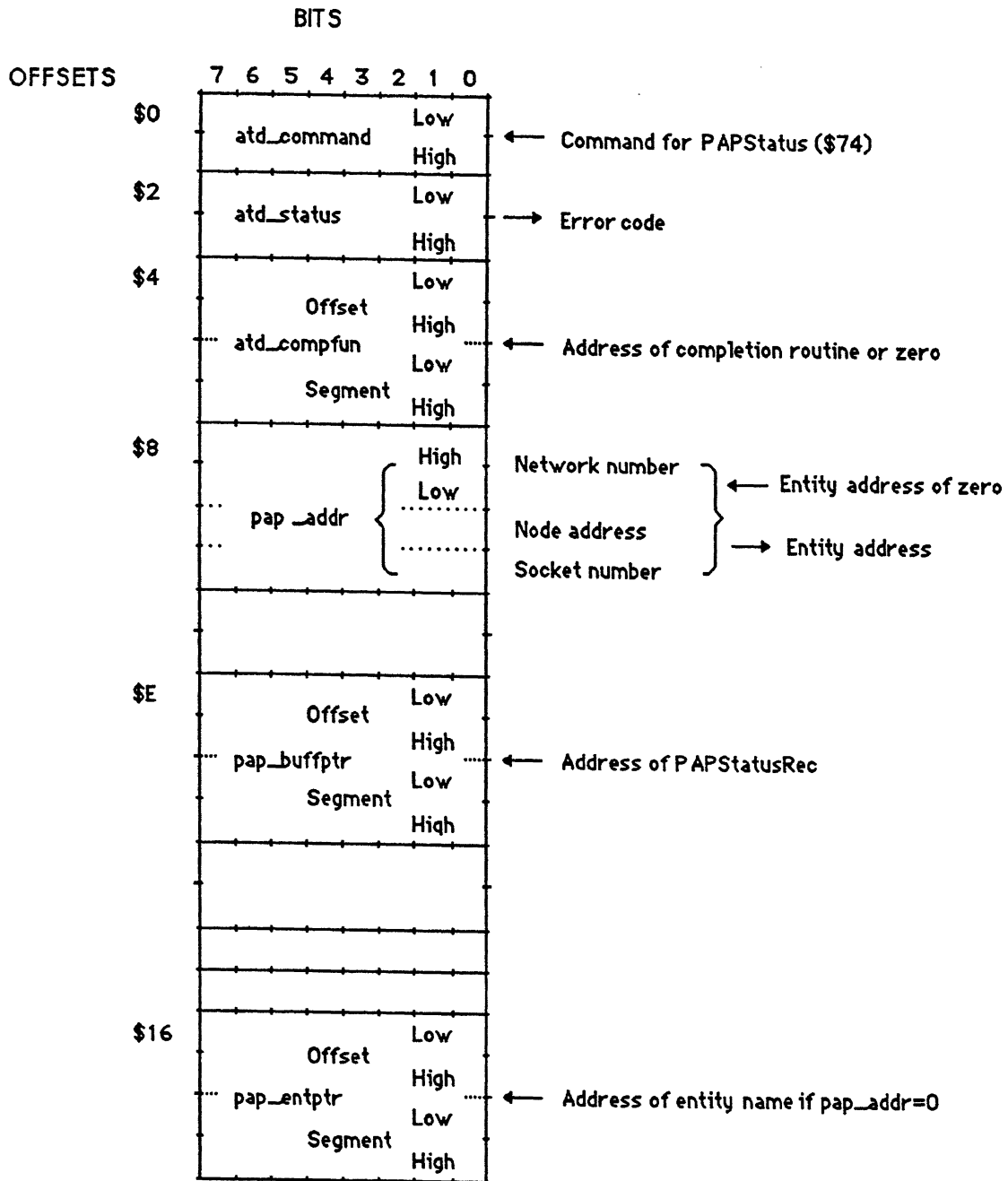


Figure 9-8. PAPStatus Parameter Block

PAPXStatus (\$7C)

The PAPXStatus call is made by a workstation to obtain the current status of a printer entity on the network.

The required inputs are a valid network address in the pap_addr field or a pap_addr field of zero and a pointer to the printer entity's name (in the form of an EntityName), which is passed in the pap_entptr field. Also required is a buffer for placing the status returned by the printer entity, which is passed in the pap_buffptr field.

Upon successful completion, the status record at the pap_buffptr field (as described earlier in "Protocol Parameter Structures" in this chapter) will have the current printer-dependent status information and the pap_addr field, if zero on input, will be filled in by the driver.

The only difference between this command and the PAPStatus command is that the user may specify the retry and interval time.

Parameter usage and error returns for the PAPXStatus command are as follows:

Parameter Usage

Input

--> atd_command	PAPXStatus
--> atd_compfun	address of completion routine or 0
--> pap_addr	entity address or zero
--> pap_eof	period of interval timer
--> pap_srefnum	number of retries to be made
--> pap_buffptr	segment:offset of PAPXStatusRec
--> pap_entptr	segment:offset of EntityName

Output

<-- atd_status	error code
<-- pap_addr	entity address

Error Returns

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
NBP_NOTFOUND

Figure 9-9 illustrates the parameter block for the PAPXStatus command.

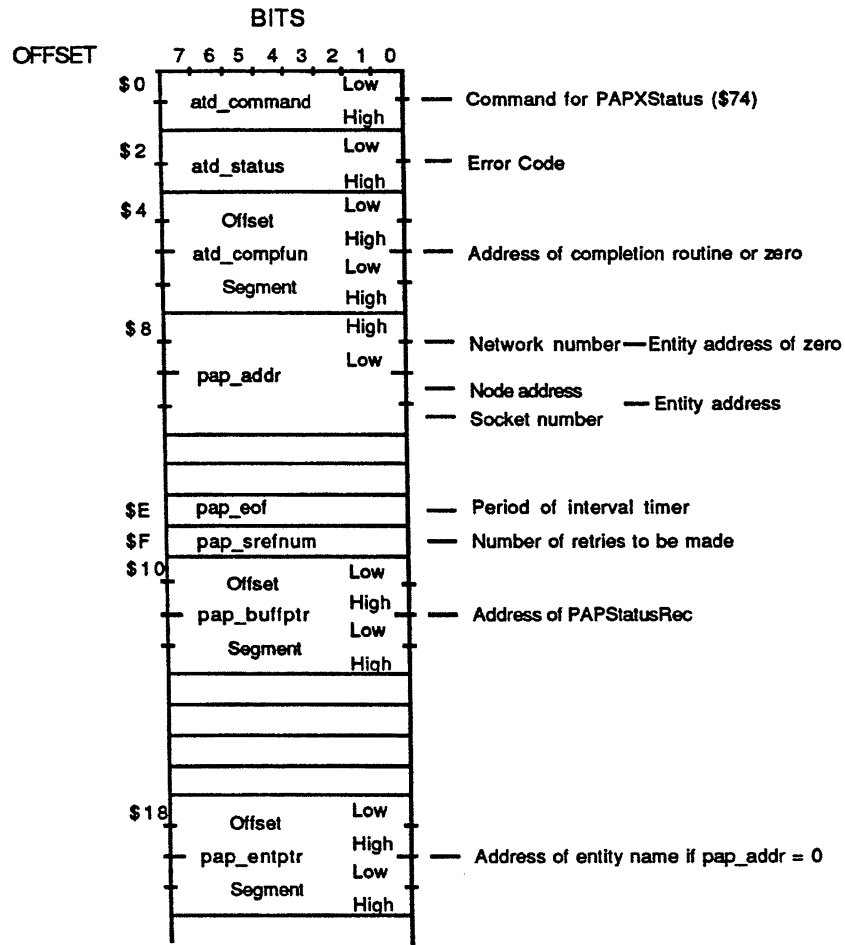


Figure 9-9. PAPXStatus Parameter Block

Chapter 10

AppleTalk Session Protocol (ASP) Commands

Introduction

The AppleTalk Session Protocol (ASP) commands create and maintain a session (a number of related transactions) between a server and a workstation.

ASP is built upon an asymmetrical model of one workstation node sending requests for service to a dedicated server node. The current version of the driver (Version 2.0) provides only workstation ASP support. This chapter describes the ASP Workstation commands. The following list contains the commands which are included with the ASP Workstation (ASPWks) protocol group:

ASPGetParms	(\$50)
ASPCloseSession	(\$51)
ASPCancel	(\$52)
ASPGetStatus	(\$5D)
ASPOpenSession	(\$5E)
ASPCommand	(\$5F)
ASPWrite	(\$60)
ASPGetAttention	(\$61)

In all cases, a return value of zero (0) means that no errors have yet occurred, and a negative number signifies some error.

For calls that are made asynchronously, it is very important that the caller not modify the parameter blocks passed to ASP until the command has been completed. The application may monitor the status word of the passed parameter structure for completion. Until the call is complete, this status word will have a positive value. Upon completion, that word will be zero (0) if there were no errors, or a negative number if there was an error.

Protocol Parameter Structures

The following structure is used by the workstation (client) end of the ASP in the interface to the AppleTalk Session Protocol commands.

Figure 10-1 illustrates the protocol parameter structure used for the Workstation interface to ASP.

AppleTalk PC Card and Driver Preliminary Note

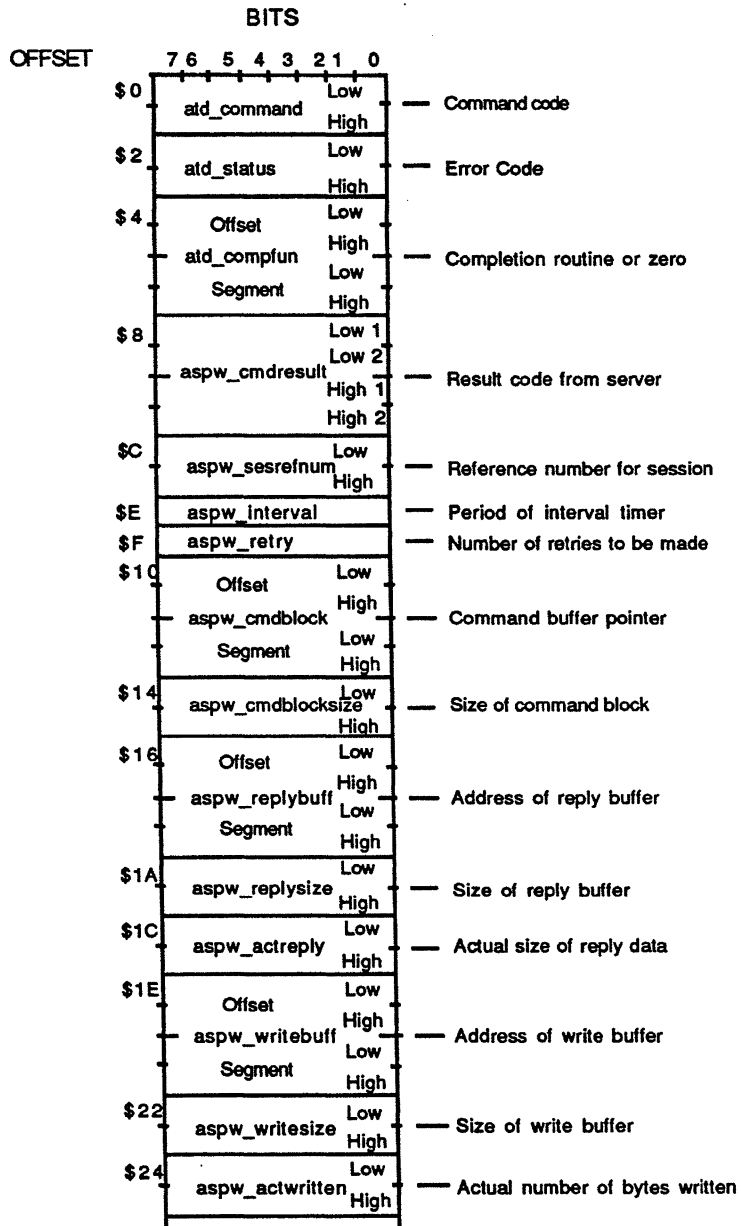


Figure 10-1. Parameter Block for ASP Workstation Commands

ASP Commands for the Workstation

The commands described in this section are made by the workstation client of ASP. Each of the command descriptions are illustrated with a parameter block.

ASPGetParms (\$50)

The ASPGetParms command returns implementation-dependent information regarding the allowable sizes of buffers.

This command must be passed to a pointer to an ASPGetParms parameter block. The maximum size of a "command block" and the maximum size of a reply or of write data are returned in the fields asp_maxcmdsize and asp_quantum.

Figure 10-2 illustrates the parameter block for the ASPGetParms command.

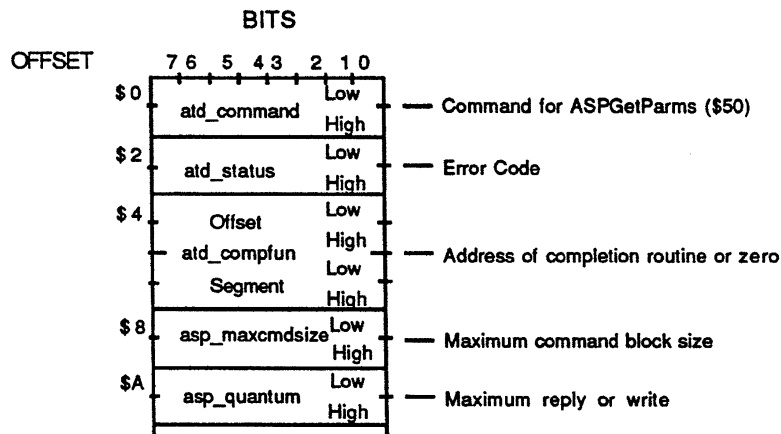


Figure 10-2. ASPGetParms Parameter Block

The error returns for the ASPGetParms command are as follows:

NOERR
ATNOTINITIALIZED

ASPCloseSession (\$51)

The ASPCloseSession command terminates an open session.

ASPCloseSession takes as input the session reference number (returned by either *ASPGetSession* or *ASPOpenSession*), and is made by the workstation client of ASP. For this command, DS:BX points to the ASPWksParams parameter block.

Figure 10-3 illustrates the parameter block for the ASPCloseSession command for the workstation.

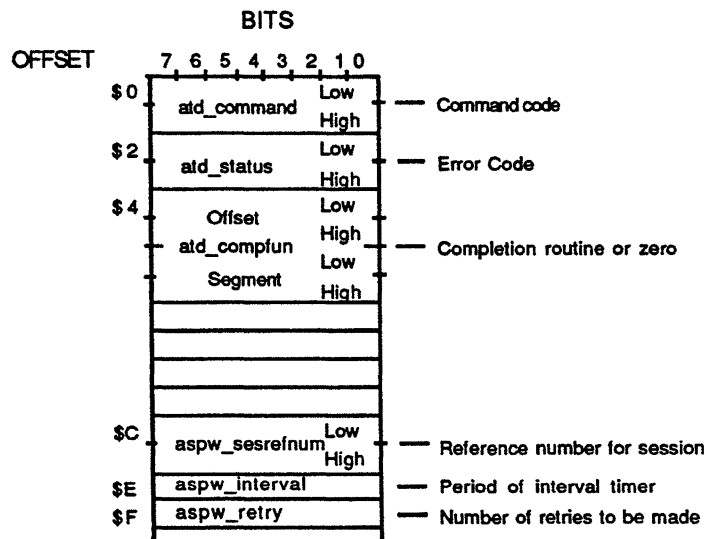


Figure 10-3. ASPCloseSession Workstation Parameter Block

The error returns for the ASPCloseSession command are as follows:

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER

ASPCancel (\$52)

The ASPCancel command cancels some outstanding ASP commands.

This command is passed to the ASPWksParams parameter block. The parameter structure address, which is passed in the aspw_cmdblock field should point to the same parameter structure as the command being cancelled. If the command being cancelled had a completion routine associated with it, that routine will be called with the atd_status field set to ASP_CANCELLED.

Since ASP is a sequenced session protocol, the cancellation of some outstanding commands could effectively break the session. Therefore, only the ASPGetStatus, ASPOpenSession, and ASPGetAttention commands may be cancelled with ASPCancel.

Figure 10-4 illustrates the parameter block for the ASPCancel command for the workstation.

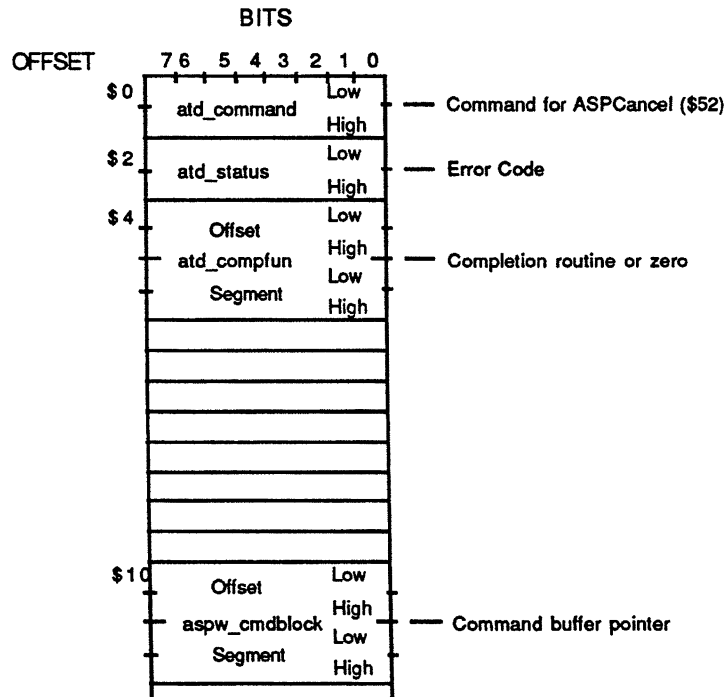


Figure 10-4. ASPCancel Workstation Parameter Block

The error returns for the ASPCancel command are as follows:

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER

ASPGetStatus (\$5D)

The ASPGetStatus command is used by the workstation to obtain the current status of a known server.

The Network Address of the server should be placed in the aspw_cmdresult field of the passed parameter structure. The aspw_replybuff field points to the status buffer, and the aspw_replysize field indicates its size.

Figure 10-5 illustrates the parameter block for the ASPGetStatus command.

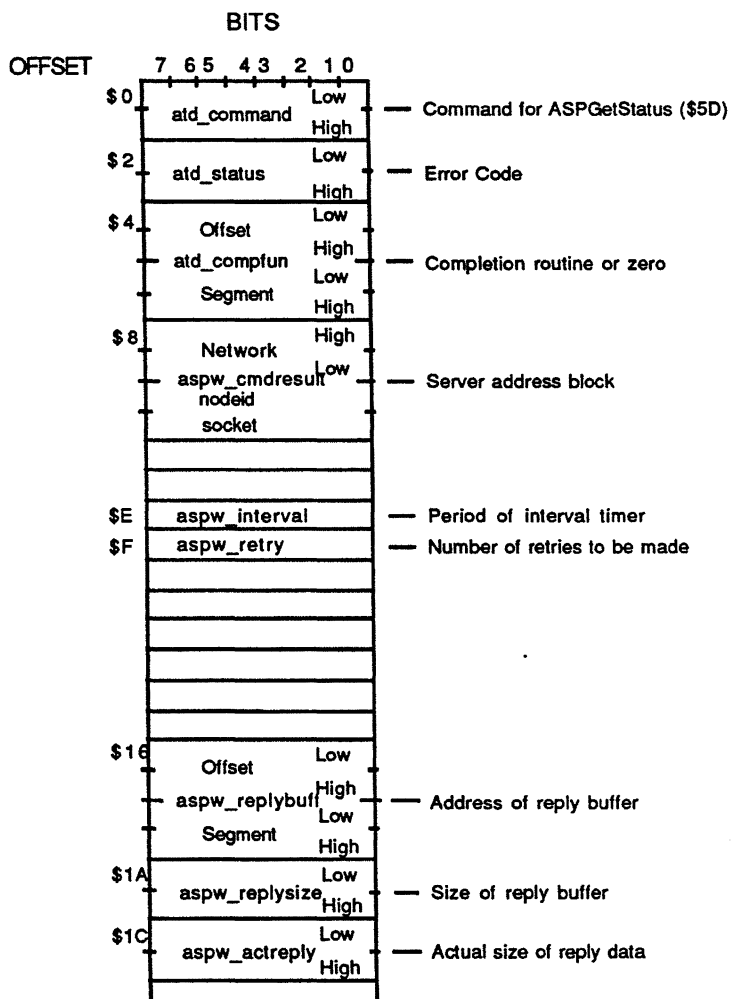


Figure 10-5. ASPGetStatus Parameter Block

The error returns for the ASPGetStatus command are as follows:

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER

ASPOpenSession (\$5E)

The ASPOpenSession command is used by the workstation to open a session with a known server. The Network Address of the server should be placed in the aspw_cmdresult field of the passed parameter structure.

Figure 10-6 illustrates the parameter block for the ASPOpenSession command.

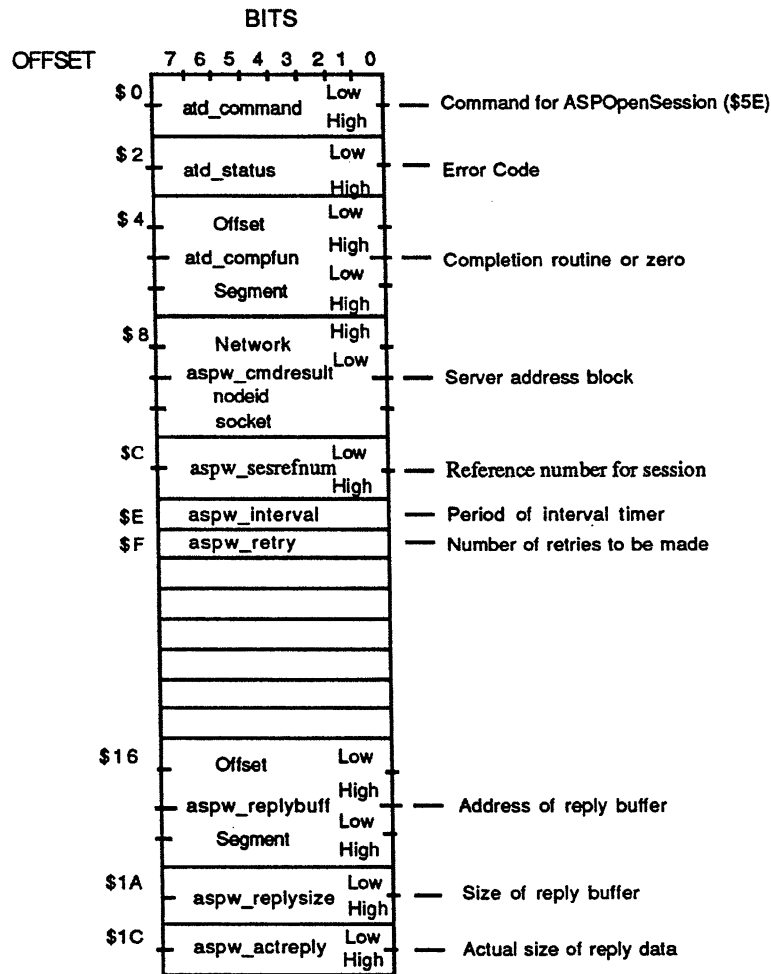


Figure 10-6. ASPOpenSession Parameter Block

The error returns for the ASPOpenSession command are as follows:

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER

ASPCommand (\$5F)

The ASPCommand command is used by the workstation on an open session to send a request of the type command to a server on an open session.

The inputs are the session reference number (SessRefNum), a command block buffer, and a reply buffer. The aspw_sessrefnum field should have the SessRefNum. The aspw_cmdblock field gives the address of the command block to be sent, and the aspw_cmdblocksize field gives the number of bytes to be sent. The aspw_replybuff field points to the reply buffer, and the aspw_replysize field indicates the size of the buffer.

Upon successful completion, the number of bytes of reply data returned will be in the updated aspw_replysize field, and the aspw_cmdresult field will have a four-byte value provided by the server.

Figure 10-7 illustrates the parameter block for ASPCommand.

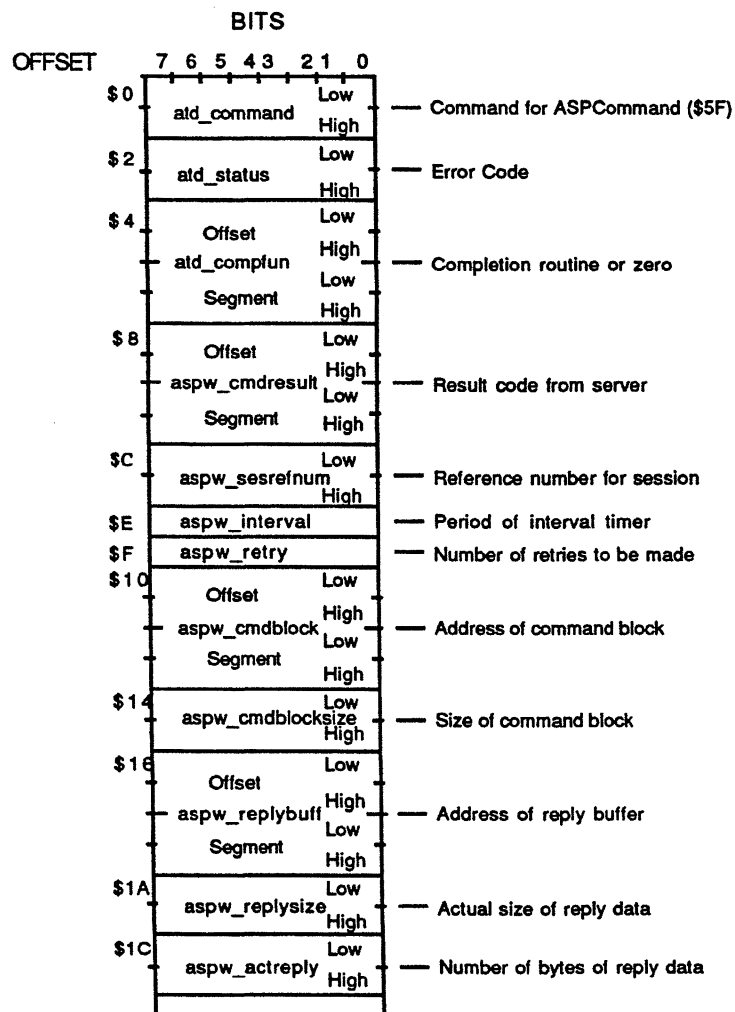


Figure 10-7. ASPCommand Parameter Block

The error returns for ASPCommand are as follows:

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER

ASPWrite (\$60)

The ASPWrite command is used by the workstation on an open session to send a request of the type write to a server.

The inputs are the same as ASPCommand described earlier, with the addition of the following: the aspw_writedata field should contain the address of the data to be sent to the server, and the aspw_writesize field gives the number of bytes.

The outputs are the same as ASPCommand with the addition of the following: upon completion, the aspw_actwritten field will contain the number of bytes successfully sent to the server.

Figure 10-8 illustrates the parameter block for the ASPWrite command.

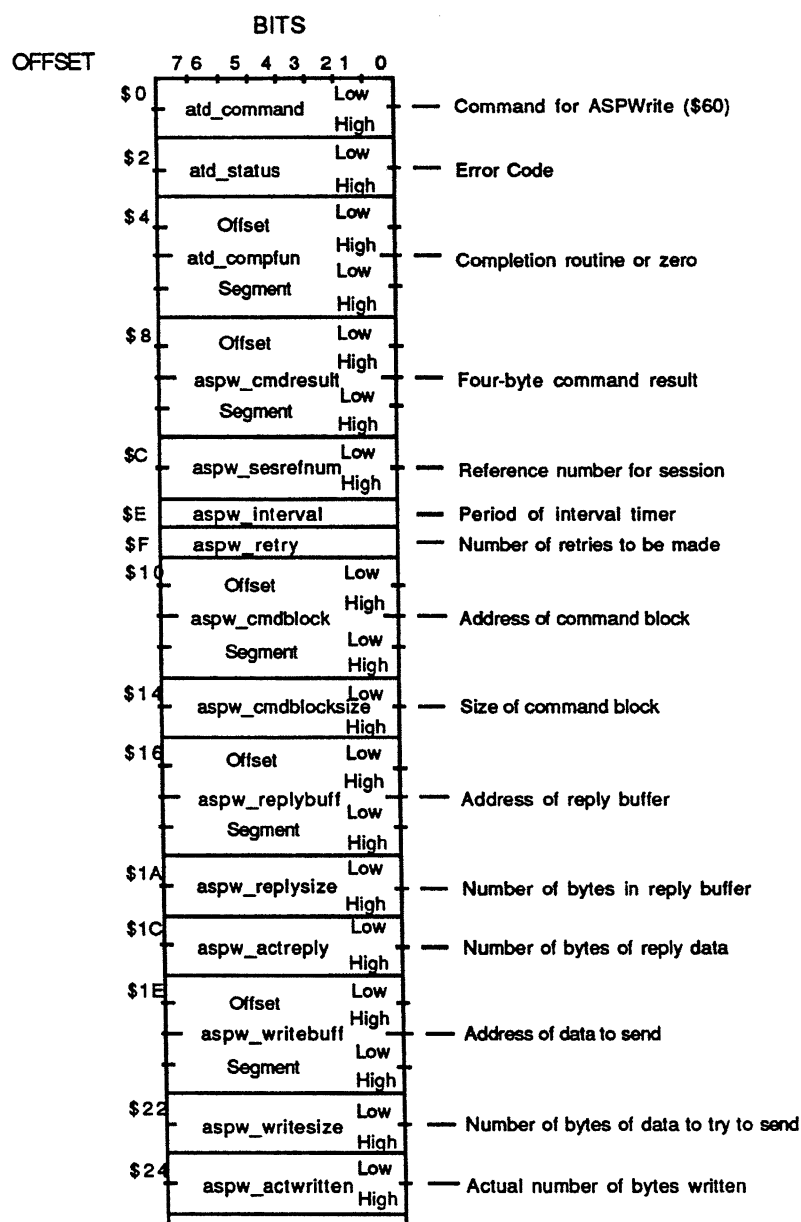


Figure 10-8. ASPWrite Parameter Block

The error returns for the ASPWrite command are as follows:

NOERR
 ATNOTINITIALIZED
 BAD_PARAMETER

ASPGetAttention (\$61)

The ASPGetAttention call is used by the workstation on an open session to listen for attention data from the server. If the server sends such data, it is sign-extended and returned in the aspw_cmdresult field.

Figure 10-9 illustrates the parameter block for the ASPGetAttention command.

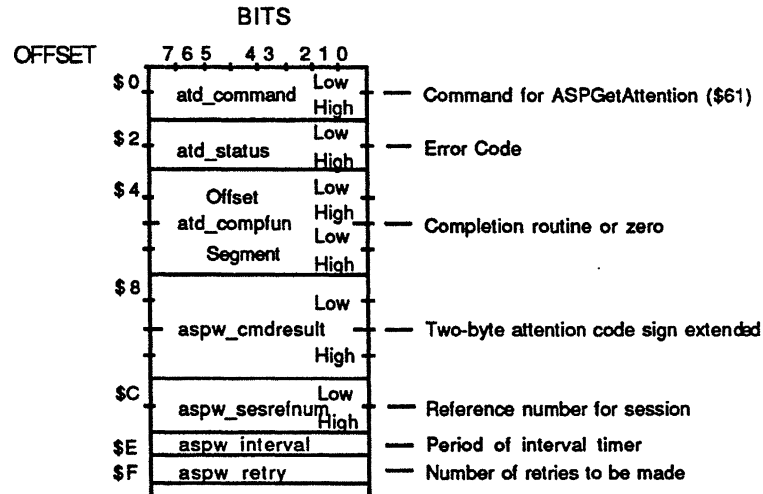


Figure 10-9. ASPGetAttention Parameter Block

The error returns for the ASPGetAttention command are as follows:

```

NOERR
ATNOTINITIALIZED
BAD_PARAMETER
ASP_DIED           (Session died - tickle time-out or Server
                   Request Close)
ASP_CANCELLED      (Session died - User call to AspCloseSession)
    
```

Appendix A

C and the AppleTalk Driver

Introduction

This appendix provides examples of how to write an application in C. The examples in this document use the Microsoft C Compiler, version 3.0.

Calling the Driver

A dispatch to the AppleTalk driver is accomplished by making sure that DS:BX points to the segment:offset of the parameter block, and executing an interrupt command to vector 60H (or the vector specified in the `/int` command when the driver was started). For example, in Microsoft C, the driver can be dispatched with the following routine in the small-memory model:

```
#include <dos.h>
int dispatch(PBlock)
{   char *PBlock;
    union REGS regs;
    regs.bx = (unsigned)PBlock;
    return (int86(0x60, &regs, &regs))
}
```

In the large-memory model, the DS register must also be set. Be sure to save the register before the `int 60H` call, and to restore the register afterwards. To do this, use the `int86x` call in Microsoft C to do this.

```
#include <dos.h>
int dispatch(PBlock)
{   char *PBlock;
    struct SREGS segs;
    union REGS regs;
    regs.bx = (unsigned)PBlock;
    segs.ds = (unsigned)(PBlock >> sizeof(unsigned))
    return (int86x(0x60, &regs, &regs, &segs))
}
```

Because the network driver returns the error status in AX, a call to `dispatch` will return an `int` value, which is the current status of the driver command. If the `dispatch` routine is called re-entrantly (that is, if a completion routine dispatches a driver command), be aware that the second invocation of `dispatch` is still using the same 512 bytes of stack.

To handle asynchronous calls correctly, the program must either poll the `atd_status` field of the parameter block, or provide a completion routine (refer to "Completion Routines and High-Level Languages" in Chapter 2).

Converting Data Types

Another set of required routines are those to convert C data types to something that the driver can understand. This conversion is especially useful for NBP and ZIP protocols, or anything that contains entity names or strings. The following sections provide examples of conversion functions for

- converting a Pascal string to a C string
- converting an entity name to three C strings
- converting three C strings to an entity name
- fetching an entity name from the Buffer returned by NBPLookup

Converting a Pascal String to a C String

The following program listing is an example of how to convert a Pascal string to a C string.

```
#include <memory.h>
char *PascalToC(dest,source)
    char *dest, *source;
{
    memcpy(dest, src+1, (unsigned)src[0]); /* Copy string data */
    dest[src[0]] = 0; /* Add the zero to end of string */
    return(src + 1 + src[0]); /* Pointer to next Pascal string */
}
```

Converting an Entity Name to Three C Strings

The following program listing is an example of how to convert an entity name to three C strings.

```
char *unpack_entity(entity, object, type, zone)
    char *entity, *object, *type, *zone;
{
    PascalToC(zone,PascalToC(type,PascalToC(object,entity)));
}
```

Converting Three C Strings to an Entity Name

The following program listing is an example of how to convert three C strings to an entity name.

```

#include <memory.h>
#include <string.h>
char *pack_entity(object, type, zone, entity)
    char *object, *type, *zone, *entity;
{
    register char *p;
    register unsigned len;
    p = entity;
    if ((len = strlen(object)) > 32) len = 32;
    p[0] = len;
    memcpy(p + 1, object, len) ;
    p += len + 1;
    if ((len = strlen(type)) > 32) len = 32;
    p[0] = len;
    memcpy(p + 1, type, len) ;
    p += len + 1;
    if ((len = strlen(zone)) > 32) len = 32;
    p[0] = len;
    memcpy(p + 1, zone, len) ;
}

```

Fetching an Entity Name From the Buffer

The following program listing is an example of how to fetch an entity name from the buffer returned by NBPLookup.

This routine will return a pointer to the Pascal string for the which entity in list (used by NBPLookup). This may be split apart by using unpack_entity. *addr will hold the network address of the entity.

```

char *NBPEextract(list, which, addr)
    char *list; int which; long *addr;
{
    register i, c = which;
    while (--c > 0)
        for (list += 5, i = 3; i--;)
            list += list[0] + 1;
    *addr = * (long *) list;
    return (list + 5);
}

```

A-04 to 09 missing in original

